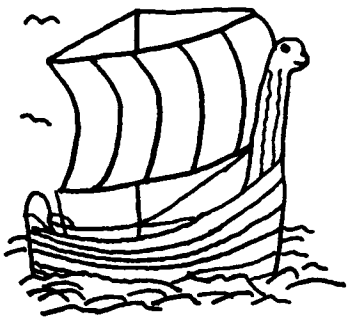


P-176

1985 Mid-Year Report

NASA Grant NAG 1-138

SAGA: A Project to Automate the Management of Software Production Systems

Principal Investigator
Roy H. Campbell

University of Illinois
Department of Computer Science
1304 W. Springfield Ave.
Urbana, IL 61801-2987.
217-333-0215

Research Assistants
Carol S. Beckman
Leonora Benzinger
George Beshers
David Hammerslag
John Kimball
Peter A. Kirslis
Hal Render
Paul Richards
Robert Terwilliger

This report details work in progress on the SAGA project during the first half of 1985.

1985 Mid-Year Report

NASA Grant NAG 1-138

SAGA: A Project to Automate the Management of Software Production Systems

Principal Investigator
Roy H. Campbell

Research Assistants
Carol S. Beckman
Leonora Benzinger
George Beshers
David Hammerslag
John Kimball
Peter A. Kirslis
Hal Render
Paul Richards
Robert Terwilliger

University of Illinois
Department of Computer Science
1304 W. Springfield Ave.
Urbana, IL 61801-2987.
217-333-0215

This report details work in progress on the SAGA project during the first half of 1985.

TABLE OF CONTENTS

APPENDICES

1. Project Overview

This report describes the current work in progress for the SAGA project. The highlights of the research in the last six months are:

- Design of ENCOMPASS, a prototype software development system, that is based on using the SAGA tools.
- Design of a Prototype Configuration Control System for SAGA.
- Completion of a prototype UNIX Pascal Language Editor; the editor includes semantic checking and incremental compiling.
- Enhancements of the SAGA Symbol Table Manager; the Manager stores the table on permanent file storage.
- Implementation of a independent String Table Manager; the string tables in the SAGA editor, and from the Symbol Table Manager will be replaced by access to this separate module.
- Enhancements of the regular right part grammar version of Olorin, the SAGA parser generator.
- Design and partial implementation of attribute-driven semantic analysis scheme in Olorin and the SAGA Editor.
- Design of a Language-Oriented Editing Language.
- Manuals and Documentation for Epos, the SAGA editor, and its associated tools.
- Partial design of a prototype SAGA executable specification language for use in software development.
- Example formal specifications using Cliff Jones "rigorous approach" and the SAGA prototype executable specification language.
- Enhancements to the ted, the proof editor; generalizations of tree editing.
- Parse tree storage in RCS. Design for integrating Diff/Undo utilities with RCS parse tree version control storage.

Appendix I contains a list of fourteen theses and papers that document the project. Five of these were produced in the last six months.

2. Introduction

The SAGA system is a software environment that is designed to support most of the software development activities that occur in a software lifecycle. The system can be configured to support specific software development applications using given programming languages, tools, and methodologies. Meta-tools are provided to ease configuration. The SAGA system consists of a small number of software components that are adapted by the meta-tools into specific tools for use in the software development application. The modules are designed so that the meta-tools can construct an environment which is both integrated and flexible. The SAGA project is documented in fourteen papers and theses, (see Appendix I.) Copies of the papers completed so far this year are included in Appendix A, C, D, E, F.

Several major steps have been taken in the last six months towards the end goal of producing practical software development systems. Several of the SAGA tools and results are being targeted to specific tasks within the NASA software acquisition lifecycle process. A prototype application of SAGA to a software development environment is being developed. The resulting environment, called ENCOMPASS, includes aspects of every phase of the software lifecycle. A complete SAGA-based Pascal editor which includes syntactic and semantic

knowledge of the language has been built as a prototype. The editor uses many of the SAGA tools including the symbol table manager. The editor interfaces to an incremental code generator to produce object code. The command language processor for the SAGA editor is being redesigned. The new user command language developed for the editor will allow the user to take advantage of the language-oriented aspects of the editor in editing commands and editing programs.

The significant results from this year's research are detailed in the following sections.

3. ENCOMPASS: A Software Engineering Environment

ENCOMPASS is an example software engineering environment being constructed by the SAGA project to support a particular model of the software lifecycle and software configurations. ENCOMPASS is based on the Vienna Development Method [Bjorner, 78], which allows the developer to start with a completely abstract specification then refine it through a number of steps into a program. The abstract specifications are based on predicate logic and predefined mathematical data types. The VDM has been used successfully on large software projects, and is suggested as a good choice for automation.

In ENCOMPASS, the software lifecycle is viewed as a sequence of developments, each of which re-uses components from the previous ones. An executable specification language is used so that programs are available for experimentation, evaluation, and validation as early as possible in the development process. By producing a running system early and often in the development process design and specification errors can be detected and corrected earlier and at lower cost.

The objects in a software system are modeled as entities which have relationships between them. An entity may have different versions and different views of the same project are allowed. ENCOMPASS supports multiple programmers and projects using a hierarchical library system containing a workspace for each programmer, a project library for each project, and a global library common to all projects. By dividing the lifecycle into a sequence of small steps, using a rigorous model for the components produced, and incorporating a hierarchical library structure, ENCOMPASS should enhance the tracking, evaluation and management of software projects. More details of the design of ENCOMPASS can be found in Appendix A.

4. SAGA Support for Configuration Control

A prototype design for a system to support configuration control has been developed and tested. The system supports reusable source and object code. The system is based on using a hierarchical file system augmented with symbolic links (that is, directory entries that refer to files stored under a new pathname from the root.) Commands are provided which allow complex directory structures to be checked in and out of RCS as single units. The system is being integrated with the hierarchical test harness and the ENCOMPASS environment. The motivation and design of the system is described in Appendix D.

5. The SAGA Editor

A number of enhancements were made to the SAGA editor: a new paging system, removal of the restriction preventing the editing of parts of a program beyond a syntax error, and the construction of a set of Pascal test programs to both check the Pascal grammar and provide regression testing of the editor. The editor has been tested and has been used to produce a Pascal editor that checks both syntax and semantics. The Pascal editor interfaces directly to the back-end of the Berkeley PC compiler.

5.1. A New Paging System

The editor's paging system was rewritten in C to permit dynamic allocation of page tables and data buffers, and to eliminate the maximum file length restriction previously required. The new routines also relaxed the restriction that the byte length of the Pascal records to be paged needed to be a power of 2; the records now may be any length.

The paging routines permit a programmer to manipulate a very large array of records stored in a file by providing random access read and write functions on a record-by-record basis, without requiring the entire file to be resident memory at once. The programmer declares a Pascal record to be used in the array, and a buffer to contain as few or as many records in memory as required. He is able to reference each record by its absolute record index in the file. The added cost of this scheme is that each record reference now requires a procedure call. This system can be very helpful if processing is required of only a few records out of the entire file, since most of the file need not ever be read into memory, saving processing time, and permitting the program to run with much less memory. The system adds considerable overhead, however, if frequent access to many records is required, and in this case may run more slowly than if the entire file were memory resident and directly accessed.

The portability of the paging system was tested when the system was transferred to an AT&T 3b2 work station. The work station runs System V Unix™, which is notably different from the BSD 4.2 Unix™ under which the paging system was developed. The porting of the software took approximately two weeks. The resulting system was then moved back to BSD 4.2 for compatibility checking, and the necessary modifications required to accomplish this task proved to be small. This indicates that the software is easily retargetable between broadly different Unix™ systems, and the portability of the editor as a whole is thus enhanced.

5.2. Editing Past Syntax Errors

Until now, the editor restricted the user to making modifications to his program only up to, but not past, the first syntax error in the program. It was necessary to repair this error before editing could be performed beyond this point. This restriction has now been removed. Editing can occur at any position in the program regardless of the number or location of existing syntax errors. If a modification is made just beyond an error, this error may prevent the parser from completely parsing the new input, but the modification will still be accepted by the editor and applied to the program. Repairing this preceeding error will permit the parser to continue with the parse.

5.3. Implementation of a Scanner Generator

In an effort to make the Mystro-based editor more generic, a scanner generator is being implemented which will produce a table-driven lexical analyzer for any editor target language. When a new target language is presented for adaptation, the adaptor will specify the lexical classes in a standard formal notation. This specification will be input for the scanner generator, which will produce tables for the deterministic finite automaton that will perform the scanning for the editor. These tables will allow the editor to be totally independent of the target language, with all language-oriented information to be loaded into the editor upon invocation. Only one copy of the executable editor code need be kept online with this system, thus greatly reducing the use of disk space. The hope is also that the resulting editor code will be less complex as it will be less dependent on the vagaries of a particular target language. This will increase the maintainability of the editor immensely.

5.4. A Test Suite for Pascal

A set of test programs has been prepared for use in checking the Pascal grammar and exercising the editor. Together, these programs cover all of the production rules in the Pascal grammar. These programs permit testing of the editor with all possible production rules that will be encountered in a given grammar. This testing is likely to uncover many of the parser problems that may be encountered using the editor with this particular language resulting in a more stable tool.

5.5. The User Interface

The editor's user interface is being revised, with consideration being given to implementing a keystroke mapping table to permit the user to assign editor commands to whatever terminal keys he wishes, thus permitting customization of the command set. The user also will be permitted to customize the command set by enabling or disabling some of the basic commands, and by writing user-defined commands based on sequences of other commands.

5.5.1. A New Editor Command Language

Currently under development is a target-independent structured editing language, tentatively called **Grendel.** The purpose of **Grendel** is to enhance the power and extensibility of the SAGA editor. The language will be the new command language for the line-mode of the editor, and must thus create a natural interface between the user and the editor. Full access to the capabilities of the editor will be provided, while still being reasonably easy to use. Some of the initial criteria for the language include:

- flexibility and extensibility;
- allowance of both structure-oriented operations and standard text-oriented operations;
- a fairly small set of primitives and combining operations, which still provide all the functionality required of either a structure or a text editor;
- user-friendliness with a uniform syntax for commands and well-defined semantics.

An initial draft of the grammar for **Grendel** is near completion, with the initial installation goal set for Fall 1985.

5.5.2. A Modular User Interface

The new editor language is just the first part of a fully modular user-interface package which will be implemented during the Fall of 1985. The three principle components of the system will be: the command interpreter, which will be a translator for the **Grendel** editor language; the display interface, which will support the windowing functions and therefore the screen mode of the editor; and the keyboard interface, which will provide direct, reconfigurable mappings of keystrokes to **Grendel** command sequences. In this way a uniform interface between the user and the editor will be created which will allow the user to tailor the editor to both himself and to the target language of the particular editor being used. All reconfigurable elements will be table-driven and will thus be loadable upon editor invocation. This would seem to provide the most useful and flexible interface to the editor.

5.6. Editor Filters

The filter command of the editor has been used to implement many functions including semantic checking, separate compilation, incremental compilation and pretty printing. The

filter command allows other programs to be executed as a coroutine from within the editor under user control. Such programs may operate on the SAGA parse tree and other files. The interface between the editor and such programs is described in Appendix G which contains a manual for "Writing Filter Processes for the SAGA Editor."

5.7. A Tree Analysis Utility: Rulecount

A program for the analysis of the parse trees produced by the SAGA editor has been implemented and tested. The program, called **rulecount,** traverses a list of parse trees and collects statistics on the number of nodes in the tree, the production rules covered, and the depth of the tree, among other values. This program will be incorporated into the editor test harness currently under development. **Rulecount** provides a useful tool for analyzing editor parse trees and thus evaluating the performance of the editor. For example, one may have a test suite for a given language for which an editor has been produced. After using the editor to create parse trees for each of the test programs, **rulecount** may be run and the coverage of the rules of the grammar for the language may be checked. In this way one could verify that the test suite does indeed use all possible language constructs. The frequency distribution of the rules can also be analyzed to ascertain possible means of improving the grammar. A test suite for the language Pascal has already been created and verified in this manner using **rulecount.**

6. Diff/Undo Version Facility

The difference system for the editor, which allows for control of multiple versions of SAGA parse tree files, has been separated into an independent program. Not only does this modification insulate the difference system from changes in the editor, but now the difference commands are executable via the editor's filter command. The filter command allows the user to execute commands which access the SAGA parse tree and which are executable from the editor command mode. This also means that the differences can also be displayed or accessed directly, without using the editor. Thus, the reusability of this tool in other SAGA tools is enhanced. Because the difference system can no longer access the editor's internal data structures, some minor restrictions have had to be imposed on the capabilities of the system. New mechanisms are being designed into the interface between the editor and the differencing system to compensate for this loss of capabilities. These mechanisms are currently being implemented.

The difference system has had a screen interface added. The differences are displayed on the screen one at a time. If a difference is too large to fit on one screen, the first part is displayed and the user can scroll the differences up or down. The old and new parts of the difference can be scrolled separately or together. When done viewing one difference, the user can go on to the next, or back to the previous one. The screen interface has also allowed the difference system to highlight the tokens that are different, so that the user can more exactly view what has changed. The screen interface makes the difference system more pleasant to use.

The version of RCS, the Revision Control System, which works with SAGA parse trees is also being modified so that RCS and the difference system can be used together. The problem that needed to be addressed was the dependence of both systems on the fields in the parse tree nodes which indicate which nodes have been modified. As the user modifies the parse tree through the editor, the modified fields in the parse tree nodes are set to indicate which nodes have been inserted and which have had neighbors deleted. Both the difference system and

RCS use these fields to find the differences between versions and then clear them to prepare for differences with the next version. Thus if a new difference base was set before the version was checked into RCS, RCS would lose some of the information that it needed. This is being solved by having the difference system "check in" a temporary version whenever the base version changes. When the user checks in the final version, the differences between the temporary versions are combined to give RCS the delta to store.

7. Symbol Table Manager

The Symbol Table Manager developed by [Richards, 84] has been modified to separate the function of the string table from the manager itself. The string table manager and symbol table manager now form two separate modules which can be used as tools by themselves. The string manager implements substantially the same interface as provided in the old symbol table, but is a stand-alone facility in order to minimize the size of the editor.

The symbol table manager has been upgraded to better support the distributed symbol tables required for type checking in separately compiled modules. Further modification to the symbol table and string table managers allows the tables to be stored permanently in files and allows access to the tables though the paging manager. The naming scheme for symbol tables has been extended to allow for file storage.

The symbol table has been used in other tools, in particular it was used to develop the Pascal editor.

8. A SAGA Pascal Editor with Semantic Checking and Code Generation

A SAGA Pascal Editor has been constructed from the SAGA Editor Epos, the Symbol Table Manager, SAGA utilites, and the Berkeley PC compiler. The Pascal editor generates intermediate code for the Berkeley PC code generator directly from the parse tree produced by the SAGA Pascal editor. When the Pascal source is modified, the utility modifies the corresponding intermediate code to reflect the changes. The intermediate code can then be compiled using the code generation passes of the PC code generator to produce VAX native code.

8.1. Semantic Checking

Semantic checking for the correctness of a Pascal program can now be accomplished within the SAGA editor. The semantic phase collects the attributes of the objects declared in a Pascal SAGA source file, and performs semantic checking to ensure the legality of the source. The semantic phase can operate either within the editor, where it provides immediate feedback to the user about semantic errors, or as a stand-alone filter, similar to a traditional compiler. The SAGA Symbol Table Manager is used to store, organize, and access the attributes collected. This ad-hoc semantic phase may in the future be replaced by a more formal, attribute-grammar based evaluator.

Utilities within the editor display semantic errors and semantic information under user control. Semantic errors are highlighted. Attributes of variables, parameters, fields of records, and elements of arrays can be displayed by selecting them using the standard editor cursor. Similarly, an error diagnosis can be obtained for a semantic error by selecting the error with the cursor. Appendix F contains more details of these utilities and displays.

8.2. Incremental Recompilation

The incremental recompilation facility utilizes the modifications-trace collected by the SAGA Make facility to control the code generator. In the first compilation of a Pascal source file, the code generator compiles the entire source, producing an intermediate code file and an object file. A new intermediate code file is produced, from which a new object file is generated. In recompilations, the incremental recompiler is guided by the Make information: walking the parse tree again, utilizing existing intermediate code where possible, and calling the code-generator to produce new code where needed. A new intermediate code file is produced, from which a new object file is generated. Further improvements to the incremental recompilation are feasible and could include reusing the relocatable binaries.

8.3. Code Generation

The code generator's input is 1) the parse tree file, 2) the symbol table, and 3) a specification of which parts of the tree to compile. Its output is binary Portable C Compiler intermediate code, such as is generated by the Berkeley Unix Pascal Compiler; this code is fed into the back-end of the Berkeley compiler to complete code-generation.

A Master's thesis was deposited this summer and details the system (called **pcg.**) The thesis is included in Appendix F.

9. The Olorin Compiler/Editor Generator

Olorin is a compiler- and editor-generator system whose goal is to produce the syntactic and semantic analysis components of a compiler or editor from regular right part LR grammars and attribute grammar specifications of programming language semantics. Several Olorin-based Epos editors have been built and work is progressing towards the automatic production of Epos editors which incorporate semantic checking.

The Olorin parser generator has been divided into two separate tools, the compiler-generator, and the editor-generator. The compiler-generator is essentially the same as the tool discussed in previous reports. The Olorin editor-generator has been restructured and substantial portions rewritten to remove the previous bias toward compiler-generation. Semantic actions have been replaced with a prototype attribute grammar scheme. The attribute grammar scheme is in the process of being extended to include maintained and constructor attributes which are discussed in Appendix C. The resulting code is in the debugging stage.

The editor-generator translates the attribute grammar into a set of parse tables and an attribute evaluation filter. The attribute evaluator is based on [Reps, 83] with extensions to support attributes over regular right part grammars and the maintained and constructor attributes. The attribute evaluator is in the final stages of coding.

The SAGA group has become the first research group to develop a general method of integrating attribute grammars with information in the surrounding software environment. These ideas have been published in Sigplan 85 [Beshers and Campbell, 85] (see Appendix C).

10. Software Specification

Research is continuing on a prototype specification language to specify and design software based upon the Vienna Development Method (VDM) [Jones, 80]. Using this method for program development, examples have been completed in detail, starting with specifications of abstract data types and ending with code in Pascal. An "Example of a Constructive Specification of a Queue" is given in Appendix B. Since the checking of the correctness of the

specifications involves repetitive rule checking. This suggests that automating this process would be reasonable. This is the case for the abstract data type examples considered. The specifications were translated from the predicate calculus used in VDM into Prolog and then proved.

Areas for further research include constructing examples of abstract programs and consideration of more complicated abstract data types than the ones already studied. Cliff Jones uses a specification language based upon predicate calculus. The main goal is to consolidate the approach of Jones to obtain a specification language which is based upon the precise use of natural language built upon a foundation of well-defined concepts.

11. SAGA Utility Functions

Over the last six months substantial effort has been expended generalizing, organizing, and documenting the standard interfaces used by SAGA tools. A standardized library of Pascal (pc compiler) to Unix™ system interfaces has been developed. Various other tools have been developed to support software development. Appendix H contains manual pages documenting these tools and interfaces.

12. Proof Management

To aid in the development of formal proofs, such as those arising in formal program verification, a proof management system is a desirable tool. A proof management system can make large, complex proofs easier to write, modify, and understand. More importantly, a proof management system also provides a means to check the validity of a proof automatically. A prototype proof management system was constructed during the year ending December 1984. Appendix E contains a paper written this spring describing the system.

Work this spring has continued on the proof management system. Some latent bugs were found and removed, additional commands were added to give the editor even greater flexibility for building and maintaining tree structures, the interfaces to the theorem provers were redone to increase modularity and reliability and to better utilize new theorem prover features. Also, we have recently connected the editor to a parsing system[1]. The addition of the parsing system allows the user to enter expressions using a more natural syntax, making the editor much easier to use, and the trees constructed much easier to read.

These changes have helped to make the system more usable; however, it has become clear that if the system is to be used in "real" situations, it must be brought out of the prototype stage. We have been slowly shifting our emphasis from the topic of proof management to the larger, more general topic of structure editing. We have come to realize that the flexibility gained by using an "unstructured" structure editor can be useful in other areas beside theorem proving and program verification. If we can develop an editor general enough (and we believe that we can), then a single extensible, customizable editor can be used to edit structures representing anything that has a tree or hierarchical structure. We have constructed prototype editors for abstract syntax trees, and for editing information trees. (An information tree is a tree where each node describes a category, and the children of that node describe subtopics in greater detail.) Additionally, preliminary work has been done on building an editor for doing program transformations.

---

[1] The parsing system was independently developed by David J. Carr and Samuel Kamin.

These prototypes have confirmed our suspicions that there are many important, powerful uses for unrestricted tree editing. Consequently, research is now underway that will result in the design of such a general purpose structure editor. We hope to design an editor that will work on tree structures in much the same spirit as our current tree editor, but will be extensible so that it can be easily customized for specific uses, without altering the editor itself. The design must also address itself to the problem of representing the tree to the user. In applications beside theorem proving, where the user is likely to move around in the tree rapidly, it is very difficult for a user to retain a sense of where she is in the tree. We anticipate that this design[2] will be done late this year.

13. Summary

We believe the SAGA project has made significant progress in this last half-year. The construction of the prototype Pascal language-oriented SAGA editor which includes semantic checking and separate intermediate code generation demonstrates the flexibility and versatility of the SAGA approach. The SAGA editor will soon be frozen and used as a foundation for generating software development systems. The new editing language and command interface will allow language-oriented program transformations to be encoded and invoked from the standard SAGA user interface. A Ph.D. thesis will document the final editor system and should be complete by January 1985.

The ENCOMPASS paper details how the SAGA system might be used to support the lifecycle of a project. A preliminary design for an executable specification language has been completed and will be documented at the end of the year. The specification language will allow a more realistic and complete experimentation with the concept of automated management of the whole cycle and will enable us to investigate methods to support reusable software.

Experiments in configuration control have confirmed the theoretical advantages and practicality of our proposed approach. This work will continue and prototype tools are being developed. The configuration control system will form a major component of the proposed ENCOMPASS environment.

Substantial progress has been on implementing a compiler and editor generator system using maintained and constructor attribute grammars. A Ph.D. thesis will document this system and should be complete by January 1985.

References

[ARM, 83] U.S. Department of Defense, *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815A-1983, U.S. Dept. of Defense, Springer-Verlag, 1983.

[Bjorner, 78] Bjorner, D., and C. B. Jones, eds., "The Vienna Development Method : The Meta-Language," Lecture Notes in Computer Science, No. 61, 1978.

[Campbell and Kirslis, 84] Campbell, Roy H., and Peter A. Kirslis, "The SAGA Project: A System for Software Development," Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, PA., Apr., 1984.

[Campbell and Lauer, 84] Campbell, Roy H., and Peter E. Lauer, "RECIPE: Requirements for an Evolutionary Computer-based Information Processing Environment," Presented at the Software Process Workshop sponsored by ACM, BCS, ERO, IEE, IEEE Computer Society, Surrey, Feb. 6-8, 1984.

[2] The design should also form the basis of an anticipated Ph.D. Preliminary examination statement.

[Jones, 80] Jones, Clifford B., *Software Development: A Rigorous Approach*, Prentice Hall International Series in Computer Science, 1980.

[Katz and Lehman, 82] Katz, R. H. and T. Lehman, "Storage Structures for Supporting Versions and Alternatives," Computer Sciences Tech. Report #479, University of Wisconsin-Madison, July 1982.

[Parnas, 77] Parnas, David L., "The Use of Precise Specifications in the Development of Software," Proc. IFIP Congress, 1977, pp. 861-867.

[Richards, 84] Richards, Paul, *A Prototype Symbol Table Manager for the SAGA Environment*, Master's Thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1984.

[Shaw et. al., 84] Shaw, R.C., Hudson, P.N., and N.W. Davis, "Introduction of a Formal Technique Into a Software Development Environment (Early Observations), " Software Engineering Notes, Vol 9, No 2, April 1984, pp. 54-79.

[Torek] Torek, C., *The Maryland Window Library,* Dept. Computer Science, University of Maryland, College Park., MD., 20742. No date given.

[Wetherell, 81] Wetherell, C.S., "Problems with the Ada Reference Grammar," ACM SIGPLAN Notices, Vol. 16, no. 9, September 1981, pp. 90-104.

ENCOMPASS: A SAGA BASED ENVIRONMENT FOR THE COMPOSITION OF PROGRAMS AND SPECIFICATIONS

Robert B. Terwilliger

Roy H. Campbell

Department of Computer Science

University of Illinois at Urbana-Champaign

Urbana,Illinois

July, 1985

Submitted to the 19th Annual

Hawaii International Conference on System Sciences

July, 1985

ENCOMPASS: a SAGA Based Environment for the Composition of Programs and Specifications

Robert B. Terwilliger
Roy H. Campbell

University of Illinois at Urbana-Champaign
Department of Computer Science
222 Digital Computer Laboratory
1304 West Springfield Avenue
Urbana, IL 61801
(217) 333-4428

Submitted to the 19th Annual
Hawaii International Conference on System Sciences

ENCOMPASS: a SAGA Based Environment for the Composition of Programs and Specifications

Robert B. Terwilliger
Roy H. Campbell

University of Illinois at Urbana-Champaign
Department of Computer Science
222 Digital Computer Laboratory
1304 West Springfield Avenue
Urbana, IL 61801
(217) 333-4428

Abstract

ENCOMPASS is an example integrated software engineering environment being constructed by the SAGA project. ENCOMPASS supports the specification, design, construction and maintenance of efficient, validated, and verified programs in a modular programming language. In this paper, we present the life-cycle paradigm, schema of software configurations, and hierarchical library structure used by ENCOMPASS. In ENCOMPASS, the software life-cycle is viewed as a sequence of developments, each of which reuses components from the previous ones. Each development proceeds through the phases planning, requirements definition, validation, design, implementation, and system integration. The components in a software system are modeled as *entities* which have *relationships* between them. An entity may have different *versions* and different *views* of the same project are allowed. The simple entities supported by ENCOMPASS may be combined into *modules* which may be collected into *projects*. ENCOMPASS supports multiple programmers and projects using a hierarchical library system containing a *workspace* for each programmer; a *project library* for each project, and a *global library* common to all projects. A prototype implementation of ENCOMPASS is being constructed on the UNIX[1] operating system using an existing revision control system and many tools developed by the SAGA project.

1. Introduction

It is widely acknowledged that software is both difficult and expensive to produce and maintain. One solution to this problem is the use of *software engineering environments* which integrate a number of tools, methods, and data structures to provide support for program development and/or maintenance[15,34,42,43]. The SAGA project is investigating both the formal and practical aspects of providing automated support for the full range of software engineering activities[2,5,7,21]. A SAGA-based software tool or environment is created by combining *standard components* which are generated by

This research is supported by NASA grant NAG 1-138.

[1]UNIX is a trademark of Bell Laboratories

meta-tools. ENCOMPASS is an example software engineering environment being developed by the SAGA group. In this paper we describe the life-cycle paradigm, schema of software configurations, and hierarchical library structure used by ENCOMPASS.

It has been suggested that *modular programming*[35] and the *top-down development* of programs[48] can help reduce the difficulty of program development and maintenance. By logically dividing a monolithic program into a number of modules we reduce the knowledge required to change fragments of the system and decrease the apparent complexity. By using *stepwise refinement* to create a concrete implementation from an abstract specification we divide the decisions necessary for an implementation into smaller, more comprehensible groups. A number of modern programming languages support modular programming[9,26,28] and environments to support modular programming have been designed[4] and constructed[41,50]. Methods to support the top-down development of programs have been devised[19,36] and put into use[37].

A *life-cycle model* describes the sequence of distinct stages through which a software product passes during its life-time[11]. There is no single, universally accepted model of the software life-cycle[3,51]. The stages of the life-cycle generate *software components* such as specifications of various forms, code written in programming languages, and many types of documentation. *Configuration management* is concerned with the identification, control, auditing, and accounting of components produced and used in software development and maintenance[1]. Configuration control systems[10,23,38] and models of software configurations[24,33] have been suggested as aids to configuration management. Life-cycle and configuration models that are understood and accepted by everyone involved can enhance communication, aid project management and increase product quality.

ENCOMPASS is a software engineering environment concerned with the construction and maintenance of efficient, validated, and verified programs in a modular programming language. The software life-cycle is viewed as a sequence of developments, each of which reuses components from the previous ones. Each development passes through the stages planning, requirements definition, validation, design, implementation, and system integration. An executable specification language is used to produce pro-

grams for experimentation, evaluation, and validation as early as possible in the development process. The components in a software project are modeled as entities which have relationships between them, and different views of the same project are allowed. The simple entities supported by ENCOMPASS may be combined into modules which may be collected into projects. ENCOMPASS supports multiple programmers and projects using a hierarchical library system containing a workspace for each programmer; a project library for each project, and a global library common to all projects.

In section two, we describe the life-cycle paradigm on which ENCOMPASS is based and in section three, we present its schema of software configurations. In section four, we describe the hierarchical library structure used by ENCOMPASS and in section five, we discuss a prototype implementation of ENCOMPASS which is being constructed on the UNIX operating system. In section six, we describe our plans for extending ENCOMPASS and in section seven, we summarize and draw some conclusions from our experience.

2. The Software Life-Cycle

ENCOMPASS is used by a programming team to construct and/or maintain a *system*, which may contain *programs* written in different languages. Modular programming techniques may be supported directly by the languages[9,26,28] or by coding conventions and/or a pre-processor[46]. A system must usually satisfy both *performance constraints*, such as speed or storage requirements, and *design constraints*, such as proper modularization and documentation. *Verification* guarantees that software components are correct and complete relative to each other, while *validation* shows that a system performs the functions desired by the *customers*[11].

It has been suggested that the *reuse* of software can significantly reduce the cost of program development[17], and systems which contain libraries of previously coded modules and/or a number of standard designs for program have been proposed[25,29]. In ENCOMPASS, any software component or group of components can be saved for later reuse in a central library. The library supports a number of concurrent projects, both accepting and supplying components for reuse in all phases of the life-cycle. ENCOMPASS supports the reuse of all the components produced in the development of a system. In

addition to source and object code, documentation, formal specifications, proofs of correctness, test data and test results can all be stored in the central library for reuse.

Figure 1 shows the proposed software life-cycle which consists of a sequence of developments. These developments might produce a series of prototypes which are used in the production of a system. In this case, each prototype would be evaluated and the results incorporated in the next stage of production. During the next stage, all the materials from the development of the prototype would be available for reuse. A sequence of developments might also produce a *family* of systems for use in different operating environments or with different optional features. In this case, all the materials from the development of the family would be available for reuse in the development of new family members. A sequence of developments might also represent what is traditionally called the *maintenance* phase of a development. A system, which has been constructed and installed, may have to be modified, corrected, or enhanced. In ENCOMPASS, this is seen as a new development, but with all the products of the previous development available for reuse. In this way ENCOMPASS supports both development and maintenance with the same methods and tools.

ENCOMPASS supports program development by successive refinement using the Vienna Development Method[19,37]. In this method, programs are first written in a language combining elements from conventional programming languages and mathematics. These *abstract programs* are then incrementally refined into programs in an implementation language. The refinements are performed one at a time and each is verified before another is applied. Therefore, the final program produced by the development and the original abstract program are equivalent. In ENCOMPASS, abstract programs may be written in the executable specification language PLEASE[44], which is an extension to the language Path Pascal[6] allowing routines and data types to be specified using predicate logic. A procedure or function may be specified using *pre* and *post-conditions* and an *invariant* for a data type may be specified.

It has been proposed that software development may be viewed as a sequence of transformations between specifications written at different *linguistic levels*[27] and systems to support similar development methodologies have been constructed[32]. ENCOMPASS supports this view of software develop-

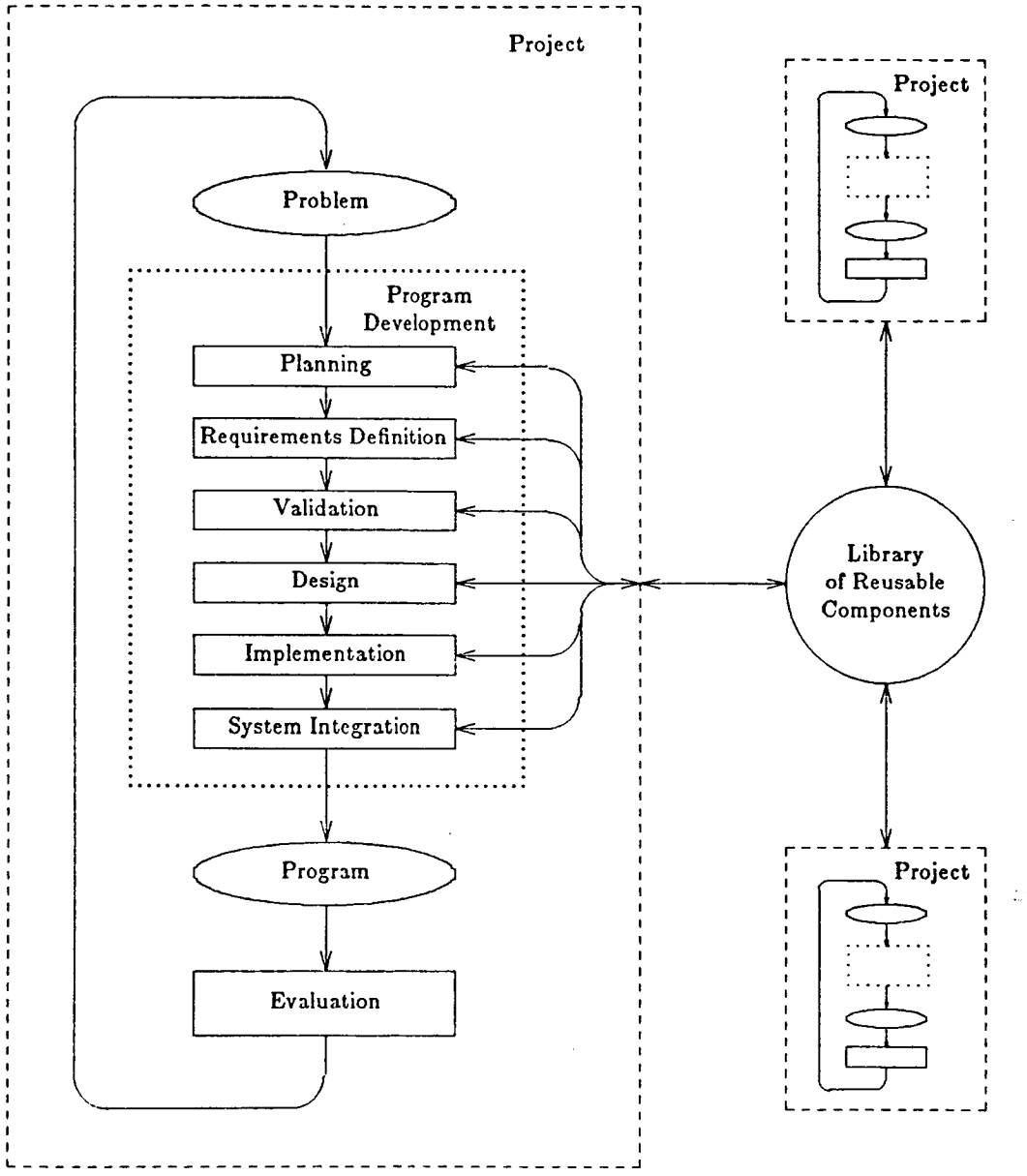


Figure 1. The Software Life-Cycle

ment by allowing abstract, predicate logic based definitions of data types or routines to be transformed into successively more concrete realizations. The use of executable specifications allows two or more linguistic levels to be run in parallel and compared for the purposes of verification or debugging.

The development steps in ENCOMPASS may be much smaller than in the traditional software life-cycle. For example, a system might go through a very large number of prototypes before delivery to the customers. Developments may also be composed hierarchically. For example, if a system is very large and complex, the production of an executable specification for the system may in itself be a complete development. If the system is composed of several major components, the production of each component might also be a complete development. By dividing the life-cycle into small steps using the mechanisms of sequential and hierarchical composition, ENCOMPASS allows each step to be smaller and more comprehensible and thereby increases management's ability to trace and control the project.

2.1. Software Development

Each development passes through the phases: planning, in which the problem is defined and it is determined if a computer solution is feasible and cost effective; requirements definition, which produces a high-level specification of the system to be produced; validation, which determines that the system described by the specification will satisfy the customers; design, in which the basic structure of the system is described; implementation, in which components of the system are constructed; and system integration, in which the components are integrated into a complete system, acceptance tests are performed, and the product is delivered. This structure is Fairley's *phased* life-cycle model[11], extended to support the Vienna Development Method and the use of an executable specification language.

The Vienna Development Method can aid in the production of correct software by allowing a system to be produced by a sequence of refinements, each of which is shown correct before proceeding further in the development. The use of an executable specification language allows each refinement to be verified by testing techniques as well as by mathematical proof. Abstract programs can also enhance the design phase by allowing experiments to be performed which influence design decisions, and the validation phase by allowing the customers to evaluate a running system early in the development process. We believe the the early validation will aid in lowering the cost of correcting errors made during requirements definition. Each phase of the development produces certain components which may be used and/or updated during the rest of the life-cycle[11].

2.1.1. Planning

In the planning phase the problem to be solved is defined and it is determined if a computer solution is feasible and cost effective[11]. Alternative solutions to the problem are considered and compared for cost effectiveness and preliminary plans and schedules for the project are created. In ENCOMPASS, these processes can be enhanced by the use of abstract programs as prototypes for experimentation and evaluation. This phase produces the two natural language documents[11]: the *system definition*, and the preliminary *project plan*. The *system definition* describes the original problem, gives justifications for the proposed computer system as a solution, and contains *acceptance criteria* which describe the standards and procedures to be used for evaluating the system. The *project plan* describes the milestones and specific products to be produced as well as the organizational structure to be used by the project. Once the problem has been defined and it is clear that a computer solution will be cost effective, a more detailed description of the system requirements is needed.

2.1.2. Requirements Definition

Requirements definition determines the functions and qualities of the software to be produced by the development[11]. This phase concentrates on the needs and desires of the customers as they affect the external system interface, rather than the internal structure of the software to be produced. This phase produces[11] the *software requirement specification*, and preliminary versions of the *users manual*, and the *software verification plan*. The *software requirement specification* precisely describes each requirement of of the software to be produced. It contains a functional specification of the system, descriptions of the external interfaces, and performance and design constraints. The *users manual* is documentation for the customers. It contains an overview of the system, tutorials on various system functions, and detailed users documentation on all system commands. The *software verification plan* describes the methods to be used in verifying that the system produced by the development satisfies the software requirement specification. Although the requirement specification describes a software system, it is not known if any system which satisfies the specification will satisfy the customers. In ENCOMPASS, we extend Fairley's phased life-cycle model to include a separate phase for customer validation.

2.1.3. Validation

The validation phase attempts to show that a system which satisfies the software requirements specification will also satisfy the customers, that is, that the requirements specification is valid. If not, then the requirements specification should be corrected before the development proceeds to the costly phases of design, implementation, and system integration. In the validation phase, the developers interact with the customers and the *system validation summary* is produced. This document describes the customers evaluation of the software requirements specification. It lists any problems encountered and the solutions agreed upon.

Traditionally, producing a correct specification is a difficult task. The users of the system may not really know what they want and they may be unable to communicate their desires to the development team. If the specification is in a formal notation it may be an ineffective medium for communication with the customers, but natural language specifications are notoriously ambiguous and incomplete. *Prototyping*[14,22], and the use of executable specification languages[20,31,52] have been suggested as partial solutions to this problems. Providing the customers with prototypes for experimentation and evaluation may increase customer/developer communication and enhance the validation process.

In ENCOMPASS, we extend Fairley's model to include software requirements specifications which are a combination of natural language and abstract programs written in PLEASE. PLEASE programs are prototypes which can be used for experimentation and evaluation, and a formal specification of a part of the system to be produced which can be used throughout the rest of the life-cycle. By providing executable programs early in the development process, errors in the requirements specification may be discovered and corrected before the internal structure of the system has been defined.

2.1.4. Design

In the design phase, the structure of the software system is defined[11]. The components of the system; their interfaces; the flow of control and data between components; and global data abstractions, structures and formats are all designed and documented. This phase produces the *software design specification*[11], which provides both a record of the design decisions made and a blueprint for the

implementation phase. This document is created in two steps: first the *architectural design specification*, and then the *detailed design specification*. In ENCOMPASS, the software design specification may contain PLEASE programs which describe the modular structure, and possibly the function, of parts of the system. These programs may be used as prototypes in experiments performed to guide the design process. They may also be used to verify parts of the design using techniques from the Vienna Development Method[19]. During the implementation phase, these PLEASE programs can be refined into programs in the implementation language Path Pascal.

2.1.5. Implementation

In the implementation phase, programming language code for the system is produced[11]. Each separately constructed module must be written, compiled, debugged, and documented. Each module must also be shown to satisfy the requirements and design specifications. In ENCOMPASS, this may be accomplished using mathematical reasoning[16,49], testing[13,18,30], technical review[47], or inspection. The use of executable specifications enhances the verification of system components using either testing or proof techniques. The executable specification for a component can be used as a test oracle against which the implementation can be compared. Since the specification is formal, proof techniques may be used which range from a very detailed, completely formal proof using mechanical theorem proving to a formal argument presented as in a mathematics text. PLEASE provides a framework for the *rigorous*[19] development of programs. Although detailed formal proofs are not required at every step, the framework is present so that they can be constructed if necessary. Parts of a project may use detailed formal verification while other, less critical parts may be handled using less expensive techniques. Once the separate components have been constructed and verified, they must be integrated and verified as a system.

2.1.6. System Integration

In the system integration phase, separately implemented modules are integrated into larger and larger units, each of which is shown to satisfy the specifications[11]. If errors are found and corrected in

a low level module, the correctness of any previously verified modules which use the low level module may have to be redetermined. This phase produces the *software verification summary*[11] which describes the results of all reviews, inspections, tests, and formal verifications which have been performed. ENCOMPASS provides tools to aid in the hierarchical integration and testing of programs. When using these tools, all modules which are used by a particular module are tested before tests of that module are begun. When the final integration has been performed the acceptance tests are performed, the product is delivered and the development is complete.

After the development has been completed a *development legacy*[11] is written. The legacy summarizes the development and provides a permanent record of what problems and solutions were encountered. This document provides both an aid to management in evaluating the effectiveness of the tools and methods used on the project, and an index to the development to be used by other developers wishing to reuse the components produced. The evaluation and reuse of components is further enhanced by the use of a configuration model to describe software components and their relationships.

3. A Model for Software Configurations

The ENCOMPASS model of software configurations is a refinement of the model presented in[21]. It is similar to the *entity-relationship* model[8] and uses the concepts of *aggregation* and *generalization*[39,40]. The model provides us with a natural way to describe software and also has a convenient representation on conventional computer systems which can be used as the basis for software engineering environments.

3.1. Entities and Relationships

An *entity* is a distinct, uniquely named component. An example of an entity is a file, which could contain the source code for a program, some test data, or an executable program. An entity may have *attributes* which describe its properties or qualities. For example, a file could have attributes such as "size", "owner", "permissions", and "modify time". An entity may be decomposed into smaller components, which may or may not be entities themselves. For example, a file might be composed of para-

graphs of text or statements in a programming language.

Two or more entities may have a *relationship* between them. For example, the entities containing the source and object code for a routine might have the relationship "compiled-from" between them. A relationship may also have attributes, for example the time the compile took place. A group of entities with a relationship between them may be abstracted into an *aggregate* entity. This entity would have entities as the values of some or all of its attributes. For example the specification[2], body, object code and load module for a group of routines might be abstracted into a single entity called a "code module". An *aggregation hierarchy* describes the way components are combined to form more and more complex structures.

A *generalization* is an abstraction which allows a number of distinct components to be grouped together into a single named component. A *generalization hierarchy* shows the way components with similar attributes are grouped into more and more general components. In our model, the set of entities which share certain attributes may be viewed as a *generic* entity. For example, the specification and body for a module might share the attributes "module name" and "type" (for example, source code, object code, test data or text). These two entities might then be grouped together into a generic component representing the source code for the module.

An entity has an internal state which may change with time. A *version* represents the state of an entity at a particular point in time. A version of an aggregate entity denotes the versions of all the entities of which it is composed. The same version of an entity may be used in many different composite entities or versions of the same aggregate entity.

3.2. Components Supported by ENCOMPASS

The aggregation hierarchy for ENCOMPASS contains three levels: *simple entities* may be combined into aggregates called *modules*, which may be collected into aggregates called *projects*. An entity

[2] In PLEASE a separately compiled module may have a specification, which describes the interface and function of the module, and a body, which contains the implementation of the module. The two are compiled as a unit to produce a single piece of object code which may be linked with other separately compiled modules to form an executable load module.

which does not have entities as the values of any of its attributes is known as a *simple entity.* An example of a simple entity is a file containing the source code for a routine with the attributes "language", "modify time", and "size". A *module* is an aggregate entity composed of other entities which are closely related or have some common property. For example, a *code module* could contain the specification, body, object code, and load module for a program. The module would have attributes specification, body, object and load with the appropriate entities as values. A *project* is an aggregate entity composed of modules. For example all the modules used in developing a program might be be grouped together into a project.

The generalization hierarchy for ENCOMPASS includes several sub-classes for both modules and simple entities. A module may be: a *code module*, which contains entities associated with the production and debugging of code; a *test module*, which contains materials for the testing of other modules such as sets of test data and test drivers or harnesses; a *proof module*, which contains entities used in the proof of a refinement; a *document module*, which contains entities used in the production of documentation; or a *history module*, which contains components used to track the history of a project. Simple entities may be: *code components*, including source code, object code, load modules and include files; *makefiles*[12], which contain instructions for compilation, linking, and testing; *test data*, such as the input or correct output from a program; *proof data*, which might be input for a mechanical theorem prover; and *document data*, such as input to text processing programs.

3.3. Views

A *view* is a mapping from names to components. A project under development has a distinguished *base view* which describes the entities of the system being designed and the primitive relationships between these entities. Other views of the project are produced from this base view by selecting, and possibly renaming, certain entities with particular attributes. For example, the development and quality assurance teams may have different views of the software system being developed by the project. The development team may use a view of the system which includes all the specifications and software being developed. However, the quality assurance team may have a different view which contains the

specifications, executable code and, in addition, the test cases. Views may be used to abstract the phases of the project corresponding to planning, requirements definition, validation, design, implementation, and system integration. Views may be used to identify a slice of the software being developed, for example, in order to restrict the activities of a programmer to a particular group of modules. Views may also be constructed to represent the effect of a modification on the rest of a system. In ENCOMPASS, access to components is controlled through the use of views and a hierarchical library structure.

4. Library Structure

Figure 2 shows the library structure used by ENCOMPASS which contains a *workspace* for each programmer, a *project library* for each project, and a *global library* common to all projects. Each programmer controls his own workspace while each *project leader* controls the library for his project and the *librarian* controls the global library. All components which are accessed by more than one programmer reside in the project or global libraries where they are controlled by either the project leader or the librarian.

A programmer accesses the components he is working with through his workspace. The workspace may actually contain these components, or it may reference components in the project or global libraries through a view. A workspace may reference the *working copy* of an components or a *version* fixed at some earlier point in time. The project library contains components that must be available to all the personnel on a particular project, and can aid the project leader in controlling and monitoring the development. The project leader controls the components in the project library by controlling access and the views into the library.

For example, a component containing the specification and body for a module might reside in the project library. Assume two programmers are working on the module. Programmer A is assigned the task of writing a specification for the module. Therefore he may access the working copy of the specification from his workspace, but he has no access to the body for the module. Programmer B is assigned the task of writing the body from the completed specification. Therefore his workspace contains references to a fixed version of the specification and the working copy of the body.

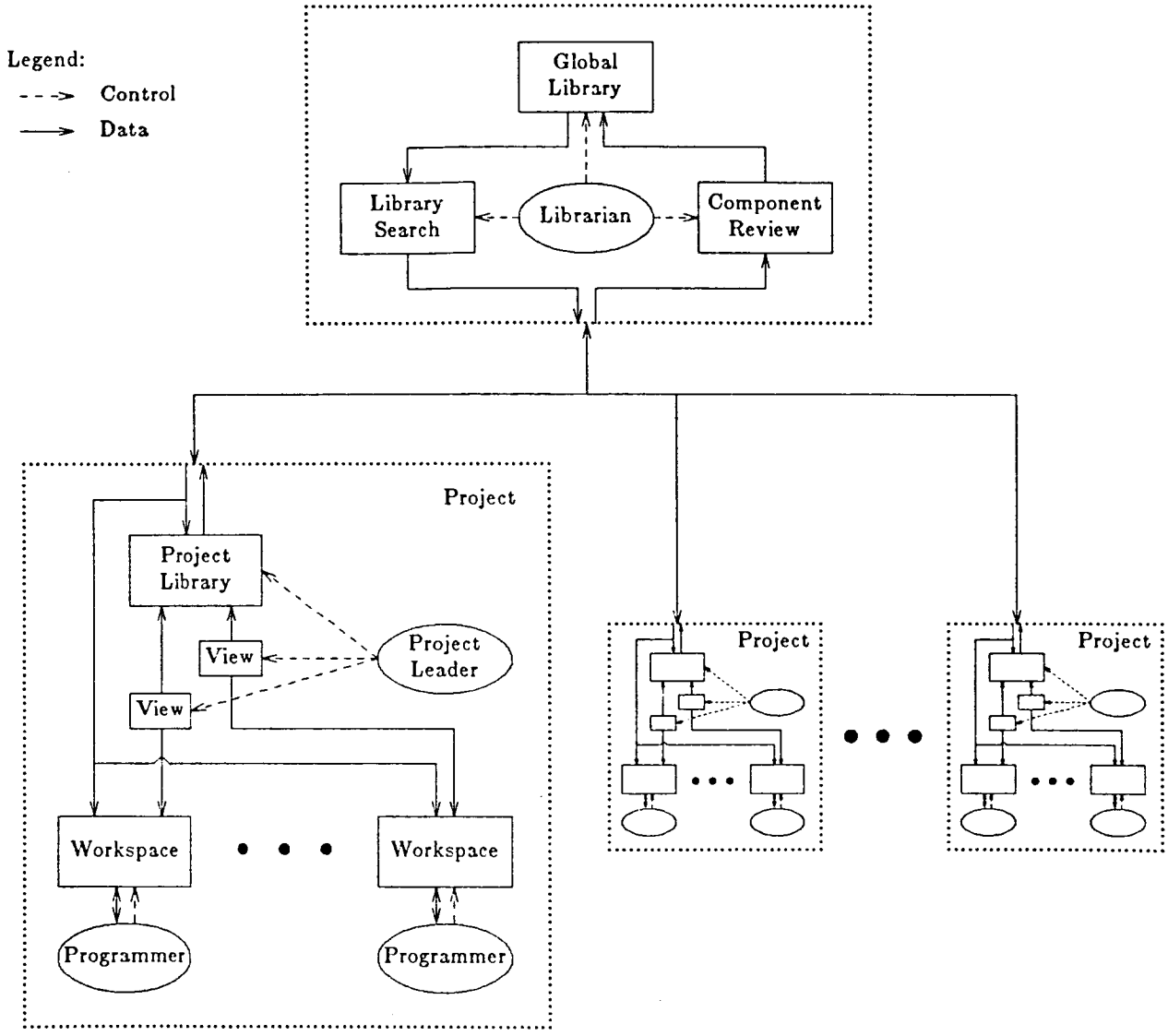


Figure 2. ENCOMPASS Library Structure

The global library contains components available for reuse on all projects and is read-only to all but the *librarian*. The librarian controls which components will be saved for reuse and how they will be available. When a project leader feels that a component may be useful for reuse on other projects he

submits it to the librarian who performs a *component review* to determine if the component meets the minimum standards for correctness, reliability, documentation, and generality. If the component meets these standards then the librarian must decide how to index the component for later retrieval. Each component available for reuse is associated with a number of *key words* which describe its structure, function and quality[3]. Components in the library may be accessed either individually or in groups. To search the library for components that may be useful, a programmer uses simple retrieval tools, specifying the key words in which he is interested using a regular expression. The tool returns a list of components, each of which is associated with the key words he specified. The programmer may then create a reference to or copy of any components which are of interest in his workspace and examine them in more detail.

For example, suppose a programmer needs a verified module which implements a stack of strings. By searching the library on the key words "stack" and "verified" he might discover that a verified module implementing a stack of integers existed in the global library. Assuming he had the proper access permissions, he could then make a copy of this module in his workspace and modify it to implement a stack of strings. The programmer may be able to reuse more than just the source code for the module. The proof data and any associated documentation could also be retrieved, modified, and reused in the new development.

5. Implementation

A prototype implementation of ENCOMPASS is being constructed on a Vax running BSD 4.2 UNIX. ENCOMPASS is designed to be an extension of the UNIX environment, so standard software tools can be used. ENCOMPASS currently encorporates standard editors, text processors, compilers, linkers and many other tools. Language-oriented tools for PLEASE are being constructed with the SAGA meta-tools. For example, a language-oriented editor for PLEASE is created from a BNF description of the language. Other language-oriented tools being constructed include an interactive tool to

[3] For example a module might have met technical review standards, be well tested, be proven by a period of use, or possibly even be formally verified with respect to its specification.

transform PLEASE programs into executable form and a verification condition generator.

The configuration control tools and the hierarchical library structure are implemented using a representation of our configuration model on the UNIX file system[21]. The representation uses files to represent simple entities, directories to represent modules and projects, and symbolic links[4] to represent complex relationships. For example, a directory representing a module may contain files representing simple entities such as the specification of the module, the body of the module, the object code, and possibly the load module. A number of tools have been written which use the underlying directory structures. For example, complex entities can be moved and copied as single units. A version of any entity can be saved using the RCS revision control system[45]. For complex entities a table containing the versions of all the sub-components is stored.

The use of symbolic links simplifies the interaction of the configuration tools and existing systems components. By implementing references between modules by symbolic links, tools such as a compiler can directly access the required source needed for the compilation and existing compilers can be used in our environment without alteration. Another benefit of the use of symbolic links is that the makefile for a module only needs to search the current directory for source dependencies. Therefore, the makefile can use pattern matching techniques to access all the relevant files in a module and does not have to be rewritten every time the modularization of the program is changed.

The workspaces and libraries are implemented as directories, which are owned by the person who controls them. These directories contain sub-directories, files and symbolic links with the meanings given above. Views are implemented as directories containing symbolic links. References from workspaces, through views, to components in the project and global libraries are implemented as chains of symbolic links. Views are created and modified by csh[5] scripts which are saved and run by project leaders. If a view references a particular version of an entity, rather than the working copy, the version is checked out of RCS into a special area of the library when the view is created. This structure has

[4] A symbolic link contains the name of the file to which it is linked. Symbolic links may span file systems and may refer to directories. The file to which the link refers need not exist at the time the link is created.

[5] Csh is a command interpreter on UNIX which supports many of the features found in modern programming languages. A sequence of shell commands may be saved and run as a program.

been used to support PLEASE, Path Pascal, C, Pascal and csh programs.

6. Future Work

Although ENCOMPASS is independent of the language used for development, currently all the language-oriented tools are being constructed for PLEASE and Path Pascal. We plan to apply our executable specification method to ADA and create the language-oriented tools to support it. We plan to extend the notion of versions used in ENCOMPASS to differentiate between sequential *revisions* and parallel *alternatives*. A revision supercedes the component from which it was created, while an alternative provides a choice between component. For example, different alternatives of a program can be maintained for use with different operating systems. Each alternative passes through a series of revisions as it evolves.

Presently the configuration control tools in ENCOMPASS can only be used on projects which follow certain conventions for directory structure. We would like to extend the implementation of ENCOMPASS to allow its use with any pre-existing directory structure on UNIX. We would also like to extend ENCOMPASS to support aggregation hierarchies of arbitrary complexity and a generalized hierarchical library structure. We plan to use ENCOMPASS to maintain itself, and to develop several new software tools. We hope that this experience will give us new insights which will be incorporated in future versions of ENCOMPASS.

7. Summary and Conclusions

ENCOMPASS is an example software engineering environment being constructed by the SAGA project to support a particular model of the software life-cycle and software configurations. In ENCOMPASS, the software life-cycle is viewed as a sequence of developments, each of which reuses components from the previous ones. An executable specification language is used so that programs are available for experimentation, evaluation, and validation as early as possible in the development process. ENCOMPASS supports the Vienna Development Method, in which a system is constructed by first producing a specification in an executable specification language and then incrementally refining it into a program in

an implementation language. Each refinement produces an executable program which may be used as a prototype system. By producing a running system early and often in the development process, design and specification errors can be detected and corrected earlier and at lower cost.

The components in a software system are modeled as entities which have relationships between them. An entity may have different versions and different views of the same project are allowed. ENCOMPASS supports multiple programmers and projects using a hierarchical library system containing a workspace for each programmer; a project library for each project, and a global library common to all projects. By dividing the life-cycle into a sequence of small steps, using a rigorous model for the components produced and used, and incorporating a hierarchical library structure, ENCOMPASS should enhance the tracking, evaluation and management of software projects.

8. References

1. Bersoff, Edward H. *Elements of Software Configuration Management.* IEEE Transactions on Software Engineering (January 1984) vol. SE-10, no. 1, pp. 79-87.

2. Beshers, George M. and Roy H. Campbell. *Maintained and Constructor Attributes.* Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments (June 1985) pp. 34-42.

3. Blum, B. I. *The Life-Cycle - A Debate Over Alternative Models.* Software Engineering Notes (October 1982) vol. 7, pp. 18-20.

4. Buxton, J. N. and V. Stenning. "Requirements for ADA Programming Support Environments, *Stoneman*", U.S. Dept. Defense, 1980.

5. Campbell, Roy H. and Peter A. Kirslis. *The SAGA Project: A System for Software Development.* Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (April 1984) pp. 73-80.

6. Campbell, Roy H. and Robert B. Kolstad. *Path Expressions in Pascal.* Proceedings of the Fourth International Conference on Software Engineering (September 1979).

7. Campbell, Roy H. and Paul G. Richards. *SAGA: A system to automate the management of software production.* Proceedings of the National Computer Conference (May 1981) pp. 231-234.

8. Chen, Peter Pin-Shan. *ER - A Historical Perspective and Future Directions.* In: The Entity-Relationship Approach to Software Engineering, S. Jajodia C. G. Davis P. A. Ng and R. T. Yeh, ed. Elsevier Science, 1983, pp. 71-77.

9. Defense, U. S. Dept. Reference Manual for the ADA Programming Language ANSI/MIL-STD-1815A-1983. Springer-Verlag, New York, 1983.

10. Estublier, J., S. Ghoul and S. Krakowiak. *Preliminary Experience with a Configuration Control System for Modular Programs.* Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (April 1984) pp. 149-156.

11. Fairley, Richard. Software Engineering Concepts. McGraw-Hill, New York, 1985.

12. Feldman, Stuart I. *Make - A Program for Maintaining Computer Programs.* Software - Practice and

Experience (1979) vol. 9, pp. 255-265.

13. Gannon, John, Paul McMullin and Richard Hamlet. *Data-Abstraction Implementation, Specification, and Testing.* ACM Transactions on Programming Languages and Systems (July 1981) vol. 3, no. 3, pp. 211-223.

14. Goguen, Joseph and Jose Meseguer. *Rapid Prototyping in the OBJ Exececutable Specification Laguage.* Software Engineering Notes (December 1982) vol. 7, no. 5, pp. 75-84.

15. Goldberg, A. Smalltalk-80: The Interactive Programming Environment. Addison-Wesley, Reading, MA, 1984.

16. Guttag, John V., Ellis Horowitz and David R. Musser. *Abstract Data Types and Software Validation.* Communications of the ACM (December 1978) vol. 21, no. 12, pp. 1048-1063.

17. Horowitz, Ellis and John B. Munson. *An Expansive View of Reusable Software.* IEEE Transactions on Software Engineering (September 1984) vol. SE-10, no. 5, pp. 477-487.

18. Jalote, Pankaj. *Specification and Testing of Abstract Data Types.* Proceedings of the IEEE Computer Software and Applications Conference (November 1983) pp. 508-511.

19. Jones, Cliff B. Software Development: A Rigorous Approach. Prentice-Hall International, Engelwood Cliffs, N.J., 1980.

20. Kamin, S. N., S. Jefferson and M. Archer. *The Role of Executable Specifications: The FASE System.* Proceedings of the IEEE Symposium on Application and Assessment of Automated Tools for Software Development (November 1983).

21. Kirslis, Peter A., Robert B. Terwilliger and Roy H. Campbell. *The SAGA Approach to Large Program Development in an Integrated Modular Environment.* Proceedings of the GTE Workshop on Software Engineering Environments for Programming-in-the-Large (June 1985).

22. Kruchten, Philippe, Edmond Schonberg and Jacob Schwartz. *Software Prototyping Using the SETL Programming Language.* IEEE Software (October 1984) vol. 1, no. 4, pp. 66-75.

23. Lampson, Butler W. and Eric E. Schmidt. *Organizing Software in a Distributed Environment.* SIGPLAN Notices (June 1983) vol. 18, no. 6, pp. 1-13.

24. ----. *Practical Use of a Polymorphic Applicative Language.* Proceedings of the 10th ACM Symposium on Principles of Programming Languages (January 1983) pp. 237-255.

25. Lanergan, Robert G. and Charles A. Grasso. *Software Engineering with Reuseable Designs and Code.* IEEE Transactions on Software Engineering (September 1984) vol. SE-10, no. 5, pp. 498-501.

26. Lauer, H. C. and E. H. Satterthwaite. *The Impact of Mesa on System Design.* Proceedings of the 4th IEEE International Conference on Software Engineering (September 1979) pp. 174-182.

27. Lehman, M. M., V. Stenning and W. M. Turski. *Another Look at Software Design Methodology.* Software Engineering Notes (April 1984) vol. 9, no. 2, pp. 38-53.

28. Liskov, Barbara, Alan Snyder, Russll Atkinson and Craig Schaffert. *Abstraction Mechanisms in CLU.* Communications of the ACM (August 1977) vol. 20, no. 8, pp. 564-576.

29. Matsumoto, Yoshihiro. *Some Experiences in Promoting Reusable Software: Presentation in Higher Abstract Levels.* IEEE Transactions on Software Engineering (September 1984) vol. SE-10, no. 5, pp. 502-512.

30. Meyers, G. J. The Art of Software Testing. John Wiley & Sons, New York, 1979.

31. Musser, David R. *Abstract Data Type Specification in the AFFIRM System.* IEEE Transactions on Software Engineering (January 1980) vol. SE-6, no. 1, pp. 24-32.

32. Neighbors, James M. *The Draco Approach to Constructing Software from Reusable Components.* IEEE Transactions on Software Engineering (September 1984) vol. SE-10, no. 5, pp. 564-574.

33. Ossher, Harold L. *A New Program Structuring Mechanism Based on Layered Graphs.* Proceedings of the 11th ACM Symposium on the Principles of Programming Languages (January 1984) pp. 11-22.

34. Osterweil, Leon J. *Toolpack - An Experimental Software Development Environment Research Project.*

IEEE Transactions on Software Engineering (November 1983) vol. SE-9, no. 6, pp. 673-685.

35. Parnas, D. L. *On the Criteria To Be Used in Decomposing Systems into Modules.* Communications of the ACM (December 1972) vol. 15, no. 12, pp. 1053-1058.

36. Ross, Douglas T. *Structured Analysis (SA): A Language for Communicating Ideas.* IEEE Transactions on Software Engineering (January 1977) vol. SE-3, no. 1, pp. 16-34.

37. Shaw, R. C., P. N. Hudson and N. W. Davis. *Introduction of A Formal Technique into a Software Development Environment (Early Observations).* Software Engineering Notes (April 1984) vol. 9, no. 2, pp. 54-79.

38. Shigo, Osamu, Yoshio Wada, Yuichi Terashima, Kanji Iwamoto and Takashi Nishimura. *Configuration Control for Evolutional Software Products.* Proceedings of the 6th IEEE International Conference on Software Engineering (September 1982) pp. 68-75.

39. Smith, John M. and Diane C. P. Smith. *Database Abstractions: Aggregation.* Communications of the ACM (June, 1977) vol. 20, no. 6, pp. 405-413.

40. Smith, John Miles and Diane C. P. Smith. *Database Abstractions: Aggregation and Generalization.* ACM Transactions on Database Systems (June 1977) vol. 2, no. 2, pp. 105-133.

41. Standish, Thomas A. and Richard N. Taylor. *Arcturus: A Prototype Advanced ADA Programming Environment.* Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (April 1984) pp. 57-64.

42. Teitelbaum, Tim and Thomas Reps. *The Cornell Program Synthesizer: A Syntax-Directed Programming Environment.* Communications of the ACM (September 1981) vol. 24, no. 9, pp. 563-573.

43. Teitelman, W. and L. Masinter. *The Interlisp Programming Environment.* Computer (April 1981) vol. 14, no. 4, pp. 25-33.

44. Terwilliger, Robert B. and Roy H. Campbell. *A Preliminary Look at PLEASE: an Executable Specification Language for Concurrent Programs.* Technical Report in Preparation, Dept. of Computer Science, University of Illinois at Urbana-Champaign (1985).

45. Tichy, Walter F. *Design, Implementation, and Evaluation of a Revision Control System.* Proceedings of the 6th IEEE International Conference on Software Engineering (September 1982) pp. 58-67.

46. Warren, Sally, Bruce E. Martin and Charles Hoch. *Experience with A Module Package in Developing Production Quality PASCAL Programs.* Proceedings of the 6th International Conference on Software Engineering (September 1982) pp. 246-253.

47. Weinberg, Gerald M. and Daniel P. Freedman. *Reviews, Walkthroughs, and Inspections.* IEEE Transactions on Software Engineering (January 1984) vol. SE-10, no. 1, pp. 68-72.

48. Wirth, Niklaus. *Program Development by Stepwise Refinement.* Communications of the ACM (April 1971) vol. 14, no. 4, pp. 221-227.

49. Wulf, William A., Ralph L London and Mary Shaw. *An Introduction to the Construction and Verification of Alphard Programs.* IEEE Transactions on Software Engineering (December 1976) vol. SE-2, no. 4, pp. 253-265.

50. Yuasa, Taiichi and Reiji Nakajima. *IOTA: A Modular Programming System.* IEEE Transactions on Software Engineering (February 1985) vol. SE-11, no. 2, pp. 179-187.

51. Zave, Pamela. *The Operational Versus the Conventional Approach to Software Development.* Communications of the ACM (February 1984) vol. 27, no. 2, pp. 104-118.

52. ----. *An Overview of the PAISLey Project - 1984.* Software Engineering Notes (July 1984) vol. 9, no. 4, pp. 12-19.

P.10

An Example of a Constructive Specification of a Queue: Preliminary Report

Leonora Benzinger

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois
June, 1985

An Example of a Constructive Specification of a Queue : Preliminary Report

Leonora Benzinger

Computer Science Dept., University of Illinois, Urbana, Illinois 61801

1. Introduction

The following is an example of the constructive specification of a queue which is done in the style of [Jones 80] using the Vienna Development Method. The basic approach is that of data type refinement. While the techniques we used are not restricted to those used by Jones, particularly with respect to the method for proving properties of the retrieve function for linked lists, the notation is consistent with his.

2. The specification of a Queue

2.1. States and types for the Queue operations

Queue = Element–list

INIT
states : Queue

ENQUEUE
states : Queue
type : Element –>

DEQUEUE
states : Queue
type : —> Element

EMPTY
states : Queue
type : —> Boolean

2.2. Pre– and post–conditions for the Queue operations

post–INIT(q,q') $\equiv$ q' = <>.

post–ENQUEUE(q,e,q') $\equiv$ q' = q || <e>.

pre–DEQUEUE(q) $\equiv$ q $\neq$ <>.
post–DEQUEUE(q,e,q') $\equiv$ q' = tl(q) and e = hd(q).

post–EMPTY(q,q',b) $\equiv$ q = q' and (b <=> q = <>).

3. A Data Refinement of a Queue in Terms of Linked Lists

3.1. A queue as a linked list

```
Queue1 = [node];
node   = record
           E : Element;
           PTR : Queue1
         end;
```

3.2. The retrieve function

The retrieve function is a function which maps the linked list representation of a queue into a list representation.

retr : Queue1 —> Queue

retr(q1) ≡ if q1 = NIL then <>
else (<q1.E> ⫴ retr(q1.PTR)).

The data type invariant for Queue and Queue1 is TRUE.

3.3. Queue1 models Queue

In order to show that Queue1 models Queue the retrieve function must map all of Queue1 into Queue and every member of Queue must be the value of some member of Queue1 under the retrieve mapping. These two conditions are stated more precisely as rules aa and ab in [Jones 80, p.187]. In addition to rules aa and ab, the pre– and post–conditions for the operations for Queue1 must imply the pre– and post–conditions for the corresponding operations for Queue for members of Queue1 mapped back to Queue by the retrieve function. These conditions are precisely stated as rules da and ra [Jones 80, p.187].

3.3.1. Rules aa and ab are satisfied by the retrieve function

aa. (∀ q1 ∈ Queue1)(∃ q ∈ Queue such that q = retr(q1)).

Proof. We use structural induction on Queue1. Suppose q1 = NIL. Then retr(q1) = <> and <> ∈ Queue.

Suppose q1 ∈ Queue1 and q1 ≠ NIL. Then retr(q1) = <q1.E> ⫴ retr(q1.PTR). By the induction hypothesis there exists q' ∈ Queue such that q' = retr(q1.PTR). Let q = <q1.E> ⫴ q'. Clearly, q ∈ Queue and q = retr(q1).

ab. (∀ q ∈ Queue)(∃ q1 ∈ Queue1 such that q = retr(q1)).

Proof. We use structural induction on Queue. Suppose that q = <>. If q1 = NIL then by the definition of the retrieve function retr(q1) = q.

Let q ∈ Queue and suppose that q ≠ NIL. It follows that q = hd(q) ⫴ tl(q) where tl(q) ∈ Queue. By the induction hypothesis, there exists q1' ∈ Queue1 such that retr(q1') = tl(q). Define q1 ∈ Queue1 as follows:

$$q1.E = hd(q) \text{ and } q1.PTR = q1'.$$

Then retr(q1) = q.

3.3.2. Specification of the operations on Queue1

To specify the operations on Queue1 in terms of pre– and post– conditions we need an extension of some of the notions introduced by Jones [Jones 80, chapter 9] for lists to linked lists. The queue operations of initialization, enqueue, and empty are straightforward to implement in terms of linked lists. A difficulty occurs in the post–condition for the enqueue operation for a queue implemented on linked lists. If we choose to introduce a new argument, say, tail to describe the element appended at the end of a queue, then tail must be expressed in terms of the new queue. This is because of the form of the post–condition for the enqueue operation at the previous level of abstraction (in terms of lists) is in terms of the new queue which is obtained from the old one by concatenation of a list of a single element to the end of the old queue.

This can be done by the following:

$$tail = <hd(rev(q1))> \quad \text{for } q1 \in Queue1$$

and properly extended notions of hd, rev (the reverse order on lists), and $<>$ to linked lists. If the post–condition for the enqueue operation is stated in terms of tail, it is very awkward to verify rule ra for this operation because the post–condition for the enqueue operation on lists is stated in terms of queues of lists, not "tail ends" of queues. This approach then seems to require a backtracking in the post–condition for the enqueue operation in terms of lists using the notion of tail.

We use another approach, which is to extend the notions used for lists in the post–condition for the enqueue operation of a queue implemented in terms of lists to corresponding notions for linked lists. This has the advantage of making the post–condition for the enqueue operation in terms of linked lists very similar in form to the post–condition for enqueue for queues of lists. This also makes makes rule ra reasonably straightforward to check.

3.3.3. Extension of the theory of lists to linked lists

We define the notions of head, tail, and concatenation for linked lists. By an abuse of notation, we use the same names for these notions which are defined for lists [Jones 80, chapter 9].

Let llist, llist1, llist2 be linked lists. Denote by hd the head of a linked list. It is defined as follows:

$$hd(llist) \equiv llist.E.$$

The tail of a linked list is denoted by tl. The definition is:

$$tl(llist) \equiv llist.PTR.$$

The length of a linked list is denoted by len. The definition is:

$$len(llist) \equiv \text{if } llist = NIL \text{ then } 0 \text{ else } 1 + len(tl(llist)).$$

The index operator extended to linked lists is given by:

$$llist(i) \equiv \text{if } i = 1 \text{ then } hd(llist)$$

else tl(llist)(i – 1).

The concatenation operator extended to linked lists is given by:

llist1 ⫴ llist2 ≡ the unique linked list such that:
(∀ i ∈ {1,...,len(llist1)} (llist(i) = llist1(i))) and
(∀ i ∈ {1,...,len(llist2)} (llist(i + len(llist1)) = llist2(i)).

We observe that llist ⫴ NIL = NIL ⫴ llist = llist.

3.3.4. The retrieve function has an inverse

To define <hd(llist)> where llist is a linked list, we need the inverse of the retrieve function. We observe that the retrieve function, retr, has a natural extension from Queue1 to List1, the collection of all linked lists, by defining retrieve as follows :

retr : List1 —> List

retr(l1) ≡ if l1 = NIL then <>
else (<l1.E> ⫴ retr(l1.PTR).

The next lemma proves that retr is 1 to 1 and therefore, the inverse exists.

Lemma. Let l1, l2 in List1 and assume that retr(l1) = retr(l2). Then l1 = l2.

Proof. The proof is by structural induction. Suppose l1 = NIL and l2 ≠ NIL. Then retr(l1) = <> but retr(l2) = <l2.E> ⫴ retr(l2.PTR). This contradicts the assumption that retr(l1) = retr(l2).

Next, let l1 ≠ NIL and retr(l1) = retr(l2) for some l2 in List1. Furthermore, suppose that for each linked sublist l1' of l1, if retr(l1') = retr(l2'), where l2' is a linked sublist of l2, then l1' = l2'. We note that l2 ≠ NIL since l2 = NIL implies that retr(l2) = <>, in which case retr(l2) ≠ retr(l1). Therefore retr(l2) = <l2.E> ⫴ retr(l2.PTR). We also have retr(l1) = <l1.E> ⫴ retr(l1.PTR). Since ret(l1) = ret(l2), <l1.E> = <l2.E> and retr(l1.PTR) = retr(l2.PTR). By the induction hypothesis, l1.PTR = l2.PTR. We conclude that l1 = l2.

We observe that the rules aa and ab hold when applied to linked lists. The proofs carry over by replacing queues implemented in terms of lists and linked lists by arbitrary lists and linked lists. Thus, the function retr is a 1 to 1 mapping onto the set of lists, List.

Let l in List. There exists a unique l1 in List1, by rule ab, such that retr(l1) = l. Define invretr as:

invretr(l) ≡ l1.

This definition can be restricted in a natural way to hold only for queues implemented in terms of lists and linked lists.

We are now in a position to extend the list notation to linked lists. Let l1 in List1. Then there exists (a unique) l in List such that retr(l1) = l. Assume furthermore that l1 ≠ NIL and that l1.E = e. We define the linked list formed from the element l1.E as follows:

<l1.E> ≡ invretr(<hd(l)>).

In particular, <hd(l1)> = invretr(<hd(l)>). Notice that the list in the term on the left is a linked list, while the list in the term on the right hand side of the equivalence is not a linked list.

3.3.5. States and types for the Queue1 operations

Queue1 = [node];
node = record
E : Element;
PTR : Queue1
end;

INIT1
states : Queue1

ENQUEUE1
states : Queue1
type : Element —>

DEQUEUE1
states : Queue1
type : —> Element

EMPTY1
states : Queue1
type : —> Boolean

3.3.6. Pre– and post–conditions for the Queue1 operations

post–INIT1(q1,q1') ≡ q1' = NIL.

post–ENQUEUE1(q1,q1',e) ≡ q1' = q1 || <e>.

pre–DEQUEUE1(q1) ≡ q1 ≠ NIL.
post–DEQUEUE1(q1,q1',res) ≡ q1' = q1.PTR and res = q1.E.

post–EMPTY1(q1,q1',b) ≡ q1' = q1 and (b <=> q1 = NIL).

3.3.7. The retrieve function is an isomorphism

Lemma. Let <e>, l1 ∈ List1 and suppose that len(l1) = n for some integer n > 0. Then (l1 || <e>).PTR = l1' || <e> where l1 ∈ List1 and len(l1) = n – 1.

Proof. Suppose n = 1. Then l1 = <e1> for some e1 ∈ Element. We have (l1 || <e>).PTR = (<e1> || <e>).PTR = <e> = NIL || <e>. NIL ∈ List1 and len(NIL) = 0.

Let len(l1) = n. Then l1 = <e1, e2, ..., en> where ei ∈ Element for i = 1, 2, ..., n and the ei's are not necessarily distinct. We have

(l1 || <e>).PTR = (<e1, e2, ..., en> || <e>).PTR
= <e1, e2, ..., en, e>.PTR
= <e2, ..., en, e>
= <e2, ..., en> || <e>.

Let l1' = <e2, ..., en>. We observe that l1' ∈ List1 and len(l1') = n – 1.

Lemma. Let <e>, l1 ∈ List1. Then retr(l1 ⫴ <e>) = retr(l1) ⫴ <e>.

Proof. We use induction on len(l1). Suppose that len(l1) = 0. Then l1 = NIL. It follows that retr(l1 ⫴ <e>) = retr(<> ⫴ <e>) = retr(<e>) = <> ⫴ <e> = retr(l1) ⫴ <e>.

Assume that the lemma holds ∀ l1' ∈ List1 for which len(l1') < n for some integer n > 0. Let l1 ∈ List1 and suppose that len(l1) = n and let l1.E = e'. We have

retr(l1 ⫴ <e>) = retr(<(l1 ⫴ <e>).E> ⫴ retr((l1 ⫴ <e>).PTR).

We note that l1.E = (l1 ⫴ <e>).E so that

retr(l1 ⫴ <e>) = <e'> ⫴ retr((l1 ⫴ <e>).PTR).

We can rewrite (l1 ⫴ <e>).PTR as l1' ⫴ <e> where len(l1') < n from the previous lemma. By the induction hypothesis,

retr((l1 ⫴ <e>).PTR) = retr(l1' ⫴ <e>) = retr(l1') ⫴ <e>.

It follows that

retr(l1 ⫴ <e>) = <e'> ⫴ (retr(l1') ⫴ <e>).

But from the definition of the retrieve function

retr(l1) = <hd(l1)> ⫴ retr(l1.PTR).

Therefore, retr(l1 ⫴ <e>) = retr(l1) ⫴ <e>.

Theorem. ∀ l1, l2 ∈ List1, retr(l1 ⫴ l2) = retr(l1) ⫴ retr(l2), that is, the retrieve function is an isomorphism from the set of linked lists to the set of lists.

Proof. We use induction on len(l2). When len(l2) = 0 we have

retr(l1 ⫴ l2) = retr(l1 ⫴ <>) = retr(l1).

In List we have

retr(l1) ⫴ retr(l2) = retr(l1) ⫴ <> = retr(l1).

Assume that retr(l1 ⫴ l2') = retr(l1) ⫴ retr(l2') for l2' ∈ List1 for which len(l2') < n for some positive integer n. Suppose that len(l2) = n. Then

retr(l1) ⫴ retr(l2) = retr(l1) ⫴ (<hd(l2)> ⫴ retr(tl(l2)))

= (retr(l1) ⫴ <hd(l2)>) ⫴ retr(tl(l2)).

By the induction hypothesis and the previous lemma,

(retr(l1) ⫴ <hd(l2)>) ⫴ retr(tl(l2) = retr(l1 ⫴ hd(l2)) ⫴ retr(tl(l2)).

Since len(l2) = n, len(tl(l2)) = n − 1 so that we can use the induction hypothesis with l2' = tl(l2). It

follows that

$$\text{retr}(l1 \,||\, <\text{hd}(l2)>) \,||\, \text{retr}(\text{tl}(l2)) = \text{retr}((l1 \,||\, <\text{hd}(l2)>) \,||\, \text{tl}(l2))$$

$$= \text{retr}(l1 \,||\, (<\text{hd}(l2)> \,||\, \text{tl}(l2)))$$

$$= \text{retr}(l1 \,||\, l2).$$

3.3.8. The operations on Queue1 model the operations on Queue

The next step is to show that each of the new operations on Queue1 : INIT1, ENQUEUE1, DEQUEUE1, and EMPTY1 correspond to the operations INIT, ENQUEUE, DEQUEUE, and EMPTY on Queue. For each of the operations on Queue1 we must show that both da and ra [Jones 80] hold, where da and ra are :

da. $(\forall\ q1 \in \text{Queue1})(\text{pre-OP}(\text{retr}(q1),\text{args}) \Rightarrow \text{pre-OP1}(q1,\text{args}))$.

ra. $(\forall\ q1 \in \text{Queue1})(\text{pre-OP1}(q1,\text{args})$ and $\text{post-OP1}(q1,\text{args},q1',\text{res}) \Rightarrow \text{post-OP}(\text{retr}(q1),\text{args},\text{retr}(q1'),\text{res}))$.

da. $(\forall\ q1 \in \text{Queue1})(\text{pre-INIT}(\text{retr}(q1),\text{args}) \Rightarrow \text{pre-INIT1}(q1,\text{args}))$.

Proof. The proof is immediate since pre-INIT and pre-INIT1 are both TRUE.

ra. $(\forall\ q1 \in \text{Queue1})(\text{pre-INIT1}(q1,\text{args})$ and $\text{post-INIT1}(q1,\text{args},q1',\text{res}) \Rightarrow \text{post-INIT}(\text{retr}(q1),\text{args},\text{retr}(q1'),\text{res}))$.

Proof. Since q1' = NIL we know that retr(q1') = <>.

da. $(\forall\ q1 \in \text{Queue1})(\text{pre-ENQUEUE}(\text{retr}(q1),\text{args}) \Rightarrow \text{pre-ENQUEUE1}(q1,\text{args}))$.

Proof. This follows immediately since the pre-conditions for ENQUEUE and ENQUEUE1 are both TRUE.

ra. $(\forall\ q1 \in \text{Queue1})(\text{pre-ENQUEUE1}(q1,\text{args})$ and $\text{post-ENQUEUE1}(q1,\text{args},q1',\text{res}) \Rightarrow \text{post-ENQUEUE}(\text{retr}(q1),\text{args},\text{retr}(q1'),\text{res}))$.

Proof. We have q1' = q1 || <e> and retr(q1') = retr(q1 || <e>). By the lemma of 2.3.7, retr(q1') = retr(q1) || <e>.

da. $(\forall\ q1 \in \text{Queue1})(\text{pre-DEQUEUE1}(\text{retr}(q1),\text{args}) \Rightarrow \text{pre-DEQUEUE}(q1,\text{args}))$.

Proof. Since retr(q1) $\neq$ <>, q1 $\neq$ NIL.

ra. $(\forall\ q1 \in \text{Queue1})(\text{pre-DEQUEUE1}(q1,\text{args})$ and $\text{post-DEQUEUE1}(q1,\text{args},q1',\text{res}) \Rightarrow \text{post-DEQUEUE}(\text{retr}(q1),\text{args},\text{retr}(q1'),\text{res})$.

Proof. We have q1 $\neq$ NIL and q1' = q1.PTR and res = q1.E. From the definition of the retrieve function, retr(q1) = <q1.E> || retr(q1.PTR). Then retr(q1') = retr(q1.PTR) = tl(retr(q1)). Finally, res = q1.E = hd(retr(q1)).

da. $(\forall\ q1 \in \text{Queue1})(\text{pre-EMPTY}(\text{retr}(q1),\text{args}) \Rightarrow \text{pre-EMPTY1}(q1,\text{args}))$.

Proof. This is immediate since the pre-conditions are both TRUE.

ra. (∀ q1 ∈ Queue1)(pre-EMPTY1(q1,args) and post-EMPTY1(q1,args,q1',res) => post-EMPTY(retr(q1),args,retr(q1'),res)).

Proof. We have q1 = q1' and (b <=> q1 = NIL). Since q1 = q1', retr(q1) = retr(q1'). But q1 = NIL implies that retr(q1) = <>. Therefore, b => q1 = NIL => retr(q1) = <>. Next, suppose that retr(q1) = <>. Since retr is 1 to 1, q1 = NIL => b. Therefore, b <=> (retr(q1) = <>).

4. The Realization of the Queue Object in Pascal

To realize the queue object in Pascal we need a refinement which maps the queue-like structure into a representation of the queue in terms of pointers and variables on the Pascal "heap".

```
Queuerep :: Heap: Ptr —> Noderep
        where Noderep :: ELT : Element
                         PTER : ^[Ptr].
```

A further refinement is necessary to go from the queue representation to an implementation of a queue in Pascal.

```
program queue;

type
  qptr = ^qrec;
  qrec = record
    qdata : char;
    qnext : qptr
  end; (* qrec *)

var
  head : qptr;
  tail : qptr;

function empty : boolean;
  begin
    empty := (head = nil)
  end; (* empty *)

procedure init;
  begin
    head := nil;
    tail := nil
  end; (* init *)

procedure enqueue(arrive : qptr);
  begin
    if arrive <> nil then
      arrive^.qnext := nil;
    if empty then
      head := arrive
    else tail^.nextq := arrive;
    tail := arrive
  end; (* enqueue *)

function dequeue(var head, tail : qtr) : char;
  begin
    if head <> nil then
```

```
    begin
      dequeue := head^.data;
      head := head^.nextq;
      if head = nil then
        tail := nil
    end
end; (* dequeue *)
```

References.

Jones, Cliff B., *Software Development : A Rigorous Approach*, Prentice-Hall International, Inc., London, 1980.

APPENDIX C AND APPENDIX D REMOVED

REPRINTS

END OF I P S DOCUMENT # 10905

****

TREE-ORIENTED INTERACTIVE PROCESSING WITH AN APPLICATION TO THEOREM-PROVING

David Hammerslag

Samuel N.Kamin

Roy H.Campbell

Department of Computer Science

University of Illinois at Urbana-Champaign

Urbana, Illinois

June, 1985

Tree-Oriented Interactive Processing with an Application to Theorem-Proving

David Hammerslag
Samuel N. Kamin
Roy H. Campbell

ABSTRACT

This paper describes our concept of "unstructured structure editing" and ted, an editor for unstructured trees. Ted is used to manipulate hierarchies of information in an unrestricted manner. The tool has been implemented and applied to the problem of organizing formal proofs. As a proof management tool, it maintains the validity of a proof and its constituent lemmas independently from the methods used to validate the proof. It includes an adaptable interface which may be used to to invoke theorem provers and other aids to proof construction. Using ted, a user may construct, maintain, and verify formal proofs using a variety of theorem provers, proof checkers, and formatters.

Keywords
Theorem Proving, proof management, structure editing, tree editing

1. INTRODUCTION

The manipulation and maintenance of detailed information in an organized manner is a problem in software engineering projects. This paper describes a management tool that aids the construction, modification, and maintenance of hierarchies of information. The tool has been implemented and applied to the problem of maintaining formal proofs.

The tool is based on a tree editor which organizes and manipulates hierarchies of text, program, or data. As a proof management tool, it maintains the validity of a proof and its constituent lemmas independently from the methods used to validate the proof. It includes an adaptable interface which may be used to to invoke automated proof methods. Using the tool, a user may construct, maintain, and verify formal proofs using a variety of theorem provers, proof checkers, and formatters.

1.1. Structure Editing

Our approach can perhaps best be described by the phrase "unstructured structure editing." Our editor allows constrained hierarchies of information to be edited but does not impose any restrictions on the editing process itself. We can explain our approach by analogy with syntax-oriented program editors [Cam84], [Tei81], [Fis84], and [Don80][1]. We see program editors as being of two types:

- These are the traditional editors, which are marked by flexibility but little power for editing structured data such as programs. For example, the operation *place* **begin** and **end** *around this statement* is not readily accomplished, because the editor has no notion of what a *statement* is. It is important to note that there is a lot of structure in the text, but that structure is not used until compilation time.
- The newer "syntax-directed" editors have knowledge of the structure being edited, and strive to maintain that structure at all times. These editors facilitate structure-oriented operations, but are generally characterized by inflexibility. For example, it is difficult to transform a while loop to a repeat loop, because this involves changing the type of statement and also interchanging the two components (Boolean expression and statement) of the statement; in whichever order these are done, the tree is temporarily in an inconsistent state.

We want particularly to emphasize that structure editors have taken *two* steps away from traditional editors, only one of which we feel is helpful:

(1) **Trees** are edited (or possibly a mixture of trees and text) rather than just **text.** Since programs have a natural tree structure, this is useful.

(2) The structure which the program must possess when editing is done, it must in effect possess throughout the editing process. In traditional text editors, it is the user's

[1] We have in fact produced a prototype program editor based on our ideas, which is described in section 5.3. However, most of our work has been done on proof editing. Our point in giving this example is to explain our structure editing philosophy in a more familiar setting.

responsibility to construct a program which is syntactically correct; the editor does not "look over his shoulder" as he is editing. Yet syntax-directed editors do not trust the user to construct a valid tree, and impose constraints on what the user can do to ensure that the tree is in a valid state *at all times.*

We believe, and our tree-editor to some extent demonstrates, that it is possible to move from text-editing to tree-editing without making the editing process any more constrained than it is in text editors.

Our editing approach exploits the manipulation of abstractions without imposing the constraints of a template editor. The editor manipulates the abstract structure without verifying that the detailed syntax and semantics are correct. Further tools are used to verify these details. A program editor based on our tree editor permits the creation of proper syntactic structures, but never *requires* syntactic correctness. Each node in the abstract syntax tree can be individually validated at the user's request. To change a while loop to a repeat loop, the order of the children would be reversed (a simple tree-editing operation), the parent node would be changed from *while* to *repeat* and the "node validator" would be invoked to verify its syntactic and semantic correctness.

2. APPLICATIONS TO THEOREM PROVING

We have constructed an editor, ted, for unstructured tree editing. The editor is more fully desribed in section 3. In brief, ted maintains an internal tree structure which is edited by the user via commands such as t (copy a node at another place in the tree), t* (copy a sub-tree at another place in the tree), e (edit the contents of a node), and m* (move a sub-tree to another place).

The editor was originally designed for use with proof trees. The editor is used to create proof trees, and external programs, such as automatic theorem provers, are used to certify that the tree created is, in fact, a proof tree.

2.1. An Example

As an example, consider one of the first (and easiest) theorems proved in our system, cancellation on the left in a group; that is that for all a, b, and c, $ab = ac \rightarrow b = c$. Initially we tried to prove this directly from the axioms for a group. Figure 1 depicts the tree we initially tried[2]. Unfortuately, the theorem provers being employed were not able to verify this fact in a reasonable amount of time, so the problem was decomposed into two lemmas, the children of the root in Figure 2. Each of these lemmas was verified by the theorem prover, and finally, the theorem was proved by using the two lemmas. Note that in the final tree, the proof of the theorem does not use the axioms concerning group theory, rather the supporting lemmas are assumed to be true, and the validity of the inference is checked.

[2] These figures are not meant to show how trees are displayed in our system, but rather to represent to tree structure being discussed

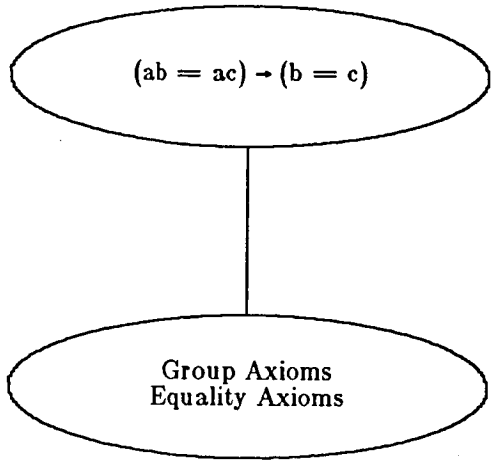


Figure 1.

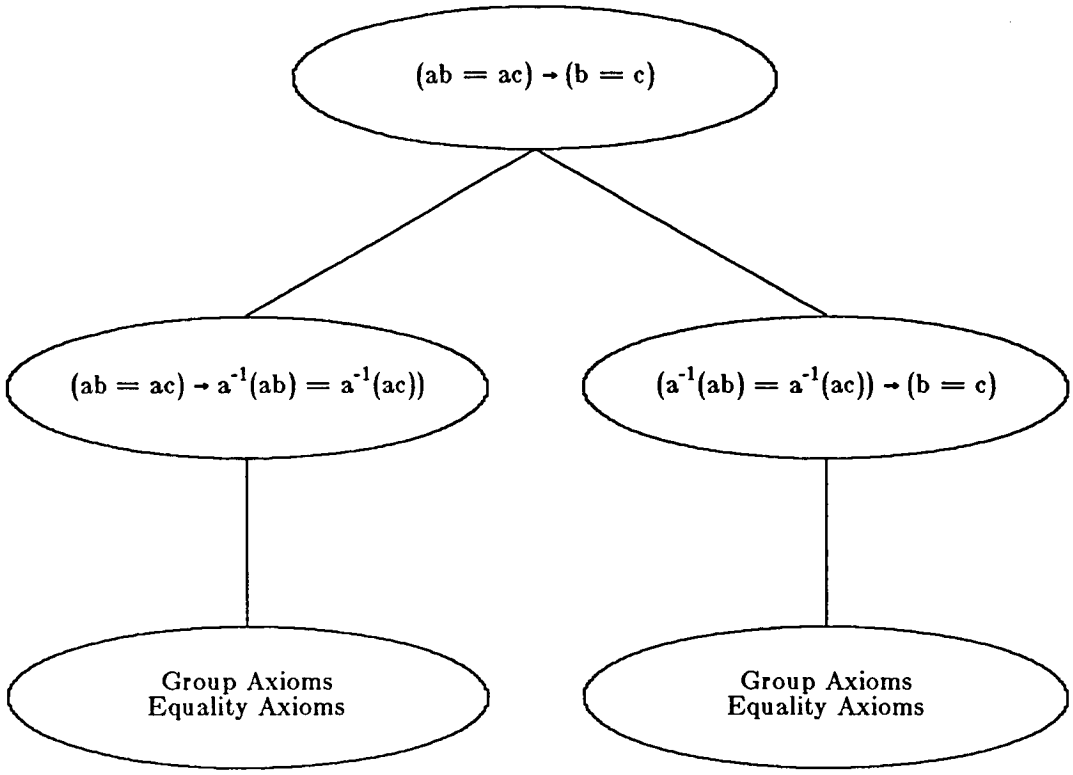


Figure 2.

2.2. Related Work

Before the editor was built, other systems (those with well-documented interfaces) were investigated [Wey77, Wey74], [Ble83, Ble73], [Gor79], [Ger79, ISI79], and [Rep84]. The systems examined can be divided into two groups: those that explicitly maintain a proof tree, and those that do not. For constructing formal proofs, a system which does not retain a proof tree leaves the user disadvantaged in a number of ways. If the user becomes lost in a proof, there is no way to retrace the steps taken. There is no way to reconstruct the proof when proving a similar or related theorem. Among systems in which an editable history is not kept are LCF [Gor79] and the UT interactive prover [Ble83,Ble73]. The UT prover makes use of user interaction to make a theorem prover faster and more efficient. However it prover provides no real flexibility: all the user can do is attempt to guide the prover down the proper path. In Edinburgh LCF, a user can write procedures which map formulas or theorems to theorems. The user defines "tactics" which are applied, under the user's control, to the proposition to be proved. While LCF provides the power of programability, it is very easy to "become lost" in a proof. The user must

often resort to naming many intermediate results just so that they can be referred to later.

Three systems which actually allow the user access to the tree being created were investigated: Affirm [Ger79,ISI79], FOL [Wey77,Wey74], and Reps' interactive proof checker [Rep84]. Although these systems allow the user to, in some sense, consult the tree, they still lack the flexibility afforded by general purpose tree editing. Affirm, a specification and verification system, includes a method for doing interactive proof development; proof tree management is emphasized in the Affirm literature. Although the user of Affirm is free to view the tree, the tree can only be manipulated by asking the system to make a specific (legal) transformation. This prevents the user from re-using parts of the tree or parts of other proof trees, or modifiying a proof by making small changes in each node. In FOL, the user inputs a rule of inference with associated parameters, where some of the parameters are usually line numbers referring to previously checked proof steps. While FOL provides a very versatile methodology for referencing previous lines, the user is still restricted to proving theorems in a bottom up manner: any proposition being used on a deduction step needs to have been previously proven. A proof checking editor has been implemented by Reps and Alpern using the Cornell Synthesizer Generator. When using the editor, the user edits program source code with imbedded assertions for the partial correctness proof in a Hoare-style logic. As the user edits the program and assertions, the editor checks the correctness of the program and proof; program fragments without valid proofs are highlighted. The interactive proof checker is very close to our approach in spirit: a user is free to move about in the tree, and proofs of supporting lemmas can be tried in any order. However, in the interactive proof checker, the proof tree is really a derivation tree for the proof in a proof language; the user has no direct access to the tree, and it can only be indirectly modified by a reparse.

3. EDITOR OVERVIEW

The prototype unstructured tree editor is written in Franz Lisp [Fod83]. The intent was to explore the uses of this form of tree editor for proof management (and possibly other uses). To make the editor easier to use, a familiar command set was desirable. Consequently, much of the editor addressing scheme, as well as the commands provided and the basic command structure, is based on that of the editor ex [Joy80]. The editor, called ted, is presented in the following manner. First a brief description of the trees being edited is given, then the method of addressing used in the editor is described. Following that the editor commands are presented.

The editor is used to create, modify and maintain proof trees in the system. Throughout the development of the editor, it was kept as general purpose as possible; that is, whenever possible, no assumptions were made concerning the interpretation to be placed on the trees being edited. As might be expected, this turned out to be difficult, and unresolved problems remain.

3.1. Tree Structure

The editor edits arbitrary trees. Each node in a tree contains an element of text and a status element. Because the editor is ignorant of the content of a node, the user is free to manipulate the node text at any time. The status of a tree node is used to indicate the consistency of the structure begin edited and access to it is restricted.

3.2. Tree Addressing

Structural addressing, similar to that used in Mentor [Don80], is used to reference individual nodes and subtrees within the tree being edited. Each node's address is dependent only upon its position in the tree at any given time. The address of a node is derived by starting at the root (denoted by "/") and, for each link traversed in getting to the desired node, appending the number of the child that that node represents to the address. In addition to using the full address of a node, a node may be specified by a base address and a combination of address offsets. There are three base addresses: the root, the current address, and address variables. The root is just "/". The previous address, the current address, and stored addresses are referenced as in ex.

In addition to base addresses, address offsets can be used to specify an address relative to some other address. The offsets are: a digit, "^", ">", and "<". A digit n, where $1 \leq n \leq 9$, addresses the nth child of a node. "^" refers to the parent of a node. ">" and "<" denote the right and left siblings of a node, respectively. For example, "1>1^" represents the second child of the current node (that is the parent of the first child of th right sibling of the first child), which may also be written simply as "2."

3.3. Editor Commands

The syntax of commands is: [tree address][command][tree address], with all parts optional. Each command has a default address (as in ex, usually ".", but "/" for w). The default command is an abbreviated print. Unless otherwise noted, the commands are the tree analogue to the corresponding ex command. There are, however, two exceptions: 1) the *'ed versions of commands refer to entire sub-trees, while the un-*'ed versions refer to single nodes; 2) in commands that have a target address (t,m), the node or sub-tree is inserted into the tree in such a way that the target address becomes the address of the new node or sub-tree. The editor commands are (default addresses in parentheses):

(†)a add a new node. † is the rightmost son of the current node.

(.)c <external prog>
invoke an external program.

(.)d, (.)d*

(.)e Call the node editor.

f [<filename>]

h [<cmd>]
help.

(.)k <letter>

(.)m<address>, (.)m*<address>

(.)p print the data at this node.

(.)p*[n]
print the "first line" of every node in the subtree down to level n (previous n if n is omitted).

q

(.)r [<filename>]

(.)s <status>
set the status of a node.

(.)t<address>, (.)t*<address>

(/)w [<filename>]

(.)=

4. A Detailed Example

In this section we present transcript of an editor session. The session shows the creation and validation of the proof tree discussed in section 2.

The following is an example of editor use for theorem proving. The example used is exactly that given in section 2.1. When the edit session starts the tree is that of figure 1. Then the axioms are deleted and the two (previously proven) lemmas are added to the tree as children of the root, giving the tree of figure 2. Finally, the proof is completed.

In the node formulas, "A" is universal quantification. It is followed by a list of quantified variables and then the formula in which they are quantified. Commands typed by the user appear in **boldface**, the text in *italics* is comment added to help explain effects of the command, all other text is output by the editor

% ted tree1
ted version 0.2

First the entire tree is displayed. Note that the status is "unproven."

⊢ p*
/ : unproven
(A (a b c) (IMP (= (* a b) (* a c)) (= b c)))

/1 : AXIOM
(A (x y z) (= (* x (* y z)) (* (* x y) z)))

/2 : AXIOM
(A (x) (= x x))

/3 : SIMP
(A (x) (= (* x id) x))

We now show the content of each of the axiom (and simplification) nodes

```
|- /1p
/1 : AXIOM
  (A (x y z) (= (* x (* y z)) (* (* x y) z)))

|- >p
/2 : AXIOM
  (A (x) (= x x))
  (A (x y) (IMP (= x y) (= y x)))
  (A (x y z) (IMP (AND (= x y) (= y z)) (= x z)))
  (A (x y) (IMP (= x y) (= (inv x) (inv y))))
  (A (w x y z) (IMP (AND (= w x) (= y z)) (= (* w y) (* x z))))

|- >p
/3 : SIMP
  (A (x) (= (* x id) x))
  (A (x) (= (* id x) x))
  (A (x) (= (* (inv x) x) id))
  (A (x) (= (* x (inv x)) id))
```

We now return to the root of the tree, delete the children just shown, and add the two lemmas as children.

```
|- ^
/ : unproven
  (A (a b c) (IMP (= (* a b) (* a c)) (= b c)))
        & & &
|- 3d
|- 2d
|- 1d
|- 1 r lemma1
```

we have now read in the first lemma, and take a look to see what's there.

```
|- p*
/1 : (c pv1)
  (A (a b c)
    (IMP (= (* a b) (* a c)) (= (* (inv a) (* a b)) (* (inv a) (* a c)))))

    /11 : AXIOM
      (A (x y z) (= (* x (* y z)) (* (* x y) z)))

    /12 : SIMP
      (A (x) (= (* x id) x))

    /13 : AXIOM
      (A (x) (= x x))

|- >r lemma2
```

Now the same for the second lemma.

```
|- p*
/2 : (c pv1)
```

```
(A (a b c) (IMP (= (* (inv a) (* a b)) (* (inv a) (* a c))) (= b c)))

  /21 : AXIOM
    (A (x y z) (= (* x (* y z)) (* (* x y) z)))

  /22 : SIMP
    (A (x) (= (* x id) x))

  /23 : AXIOM
    (A (x) (= x x))
```

Finally, we return to the root and validate the proof by invoking one of the provers, pv1.

```
├ ^
/ : unproven
  (A (a b c) (IMP (= (* a b) (* a c)) (= b c)))
       & &

├ c pv1
Skolemizing:
Pri: 6 7 10 7 [3]
------------------------------------------
Stored (Non_input + input = total): 3 + 4 = 7
1 proof, tree size: 6
PROOF:
 3 CONTRADICTION    <- (7 1)
Time (sec):    CPU: 0.45    GC: 0.0    Total: 0.45
```

The proof is successful, so the root's status has been updated to show that the node was certified by calling pv1.

```
├ p
/ : (c pv1)
  (A (a b c) (IMP (= (* a b) (* a c)) (= b c)))
```

The tree representing the completed proof is written to the current file.

```
├ w
├ q
%
```

5. DISCUSSION

This section describes work that has been done using the editor as a front-end for theorem proving and other applications to and extensions to unstructured structure editing.

5.1. Experience

The editor has been used extensively as a front-end for theorem provers. The editor was originally conceived of as a theorem prover front-end; a number of proofs have been completed

using the editor in this capacity.

Sam Kamin and Myla Archer have successfully used the editor to do proofs in group theory and category theory. David J. Carr has been a major user of the proof system, and has used it to prove the homomorphism conditions necessary to show the implementation of the tree-address data type correct with respect to a final algebra specification [Kam83] of that data type. (A further discussion of ted and proving specifications follows below.) Finally, Carol Beckman and David Hammerslag have used the system to prove (most) of the verification conditions for a program from the Basic Linear Algebra Subprogram package.

5.2. Other Applications

As the editor was designed to be application independent, it has been used in other applications. The editor was modified to manipulate trees that represent hierarchical information and to edit abstract syntax trees for a simple programming language.

5.2.1. Browse

The information tree editor, called browse, was created to allow users to peruse a hierarchy of information about programs, narrowing the class of programs under consideration as he moves down into the tree structure. Each internal node in the information tree contains information about a related group of programs; in general, a child contains information about a subset of those programs encompassed by the node's parent. Each leaf node in the tree contains information for a single program.

Except for re-writing the editor's output routines to reflect the new interpretation of node text, only minor changes were necessary (*e.g.* the *locate* command was re-written to search for nodes containing certain strings in their text). Browse was used to create and maintain a database of information concerning user contributed software.

5.2.2. FASE

Ted was originally developed to provide user-friendly access to theorem-provers, for use in program verification. As such, it is connected to a program specification system called **fase** [Kam83]. These connections, which we hope to strengthen in the future, include:

Syntax

> **fase** has been designed to allow for user-defined syntax for discussing modules appearing in the program specification. This syntax can also be used in ted nodes to state properties of the modules to be proven.

Theory

> Ted proofs inherit axioms relating to modules from the **fase** specifications. At present, this is done manually, but we expect to have it automated in the near future. In the same way, proofs of module implementations are based upon **fase** specifications and the associated formalism.

Thus, ted is really just one major component of a system which includes (executable) program specifications and program proofs. Eventually, we hope to expand this into a program development system with the inclusion of program-editing and execution facilities (based upon

the peg model), and program transformations.

5.2.3. Peg

A third use of tree editing which has been explored is program syntax tree editing. The editor, peg, allows the user to construct programs by either explicitly constructing abstract syntax trees or by filling a node with source code for the language. In the system, the user is provided with tools to transform text into an abstract syntax tree and to compress syntax trees back into text. Each node in the tree is labeled to indicate which syntactic entity the node or subtree represents. As is the case with the proof system, the editor keeps track of which nodes have been invalidated.

Although peg has only been implemented for a very restricted subset of the programming language C, we feel that it demonstrates the advantages of unstructured syntax tree editing over the syntax directed editing schemes discussed in section 1.

5.3. Conclusion

Our experience in applying our philosophy of structure editing has been favorable. We have completed and used extensively a prototype system for managing formal proofs and have experimented with other applications. We feel that as this system is made more general (that is, less dependent on the application domain) and is given a better user front end more uses will be found for unstructured structure editing.

REFERENCES

[Ble73] W. W. Bledsoe and Peter Brunell, "A Man-Machine Theorem Proving System," *Third IJCAI,* pp. 56-65 (1973).

[Ble83] W. W. Bledsoe, "The UT Interactive Prover," The University of Texas, Austin, Austin (April 1983).

[Cam84] Roy H. Campbell and Peter A. Kirlis, "The SAGA Project: A System for Software Development," *SIGLPAN Notices* **19** pp. 73-80 (May 1984).

[Don80] V. Donzeau-Gouge, G. Huet, G. Kahn, and B. Lang, "Programming Environments Based on Structure Editors: the MENTOR Experience," INRIA, France (May 1980).

[Fis84] C. N. Fischer, Anil Pal, Daniel L. Stock, Gregory F. Johnson, and Jon Mauney, "The POE Language-Based Editor Project," *SIGPLAN Notices* **19** pp. 21-29 (May 1984).

[Fod83] John K. Foderaro, Keith L. Sklower, and Kevin Layer, "The FRANZ LISP Manual," University of California Berkeley, Berkeley, California (June 1983).

[Ger79] S. L. Gerhart, D. R. Musser, D. H. Thompson, D. A. Baker, R. W. Erickson, R. L. London, D. G. Taylor, and D. S. Wile, "An Overview of Affirm: A Specification and Verification System," USC Information Sciences Institute, Marina del Ray, Ca (November 1979).

[Gor79] Michael J. Gordon, Robin Milner, and Christopher P. Wadsworth, *Edinburgh LCF,* Springer-Verlag Lecture Notes in Computer Science, No. 78, New York (1979).

[Joy80] William Joy, "Ex Reference Manual Version 3.5/2.13," Computer Science Division, University of California, Berkeley, Berkeley, California (September 1980).

[Kam83] Samuel N. Kamin, Stanley Jeffeson, and Myla Archer, "The Role of Executable Specifications: The FASE System," *Proc. IEEE Symposium on Application and Assessment of Automated tools for Software Development,* pp. 105-114 (November 1983).

[Rep84] Thomas Reps and Bowen Alpern, "Interactive Proof Checking," *Proceedings of Eleventh ACM Symposium on Principles of Programming Languages.*, pp. 36-45 (January 1984).

[ISI79] ISI Research Staff, "Program Verification: Annual Report, Fall 1979," USC Information Sciences Institute, Marina del Ray, California (1979).

[Tei81] Tim Teitelbaum and Thomas Reps, "The Cornell program synthesizer: a syntax directed programming environment," *Comm. ACM* **24** pp. 563-573 (Sept 1981).

[Wey74] Richard Weyhrauch and Arthur J. Thomas, "FOL: a Proof Checker for First-Order Logic," Stanford Univesity, Computer Science Department, Stanford, California (September 1974).

[Wey77] Richard Weyhrauch, "A Users Manual for FOL," Stanford University, Computer Science Department, Stanford, California (July 1977).

PCG: A PROTOTYPE INCREMENTAL COMPILATION FACILITY FOR THE SAGA ENVIRONMENT

Joseph John Kimball

Department of Computer Science

University of Illinois at Urbana-Champaign

Urbana, Illinois

July, 1985

PCG: A PROTOTYPE INCREMENTAL COMPILATION FACILITY FOR THE SAGA ENVIRONMENT

BY

JOSEPH JOHN KIMBALL

B.A., Creighton University, 1980

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1985

Urbana, Illinois

TABLE OF CONTENTS

Chapter 1

INTRODUCTION

A programming environment supports the activity of developing and maintaining software. New environments provide language-oriented tools such as syntax-directed editors, whose usefulness is enhanced because they embody language-specific knowledge. When syntactic and semantic analysis occur early in the cycle of program production, that is, during editing, the use of a standard compiler is inefficient, for it must re-analyze the program before generating code. Likewise, it is inefficient to recompile an entire file, when the editor can determine that only portions of it need updating.

The pcg, or Pascal code generation, facility described here generates code directly from the syntax trees produced by the SAGA syntax-directed Pascal editor. By preserving the intermediate code used in the previous compilation, it can limit recompilation to the routines actually modified by editing.

1.1 Compilation in Software Development Environments

Within the formalisms developed to aid the software lifecycle, the actual process of writing code is itself a cycle: think, edit, compile (and link-edit

if needed), test, and think again.

A software development environment provides tools for program creation and maintenance. In the traditional software development environment, the most visible tools are the editor and the compiler. The division of labor between the two is as follows. The editor is used for entering and modifying code; it is text-oriented, suitable for the entry of any type of text. As a general-purpose tool, the editor cannot provide assistance for any particular language. The compiler, on the other hand, is specific to one programming language, and does two jobs: 1) it must check the source code's syntax and semantics, to ensure that the code constitutes a legal program in the language, and 2) it must then translate that legal program into executable form. Therefore, if the compiler discovers static errors in the source file, it aborts, and the programmer must return to the editing phase to make corrections. The compiler must be run repeatedly merely to obtain error diagnoses, making checking for errors very costly [Campbell and Kirslis] [Medina-Mora and Feiler].

The more helpful of traditional environments provide an automatic facility to drive the compilation and link-editing phase, for separately-compiled programs. The 'make' program [Feldman] under Unix[1] is an example. Its knowledge is embodied in 1) a user-supplied description of the dependencies among the various files, and 2) the file system's timestamp which records when a file was last modified. Given these, Unix make can determine which files must be updated after a modification to one occurs. If a file has not been reconstructed since the files from which it is built were modified,

---

1. Unix is a trademark of Bell Laboratories.

make will recompile and re-link as needed to update the program.

In the traditional environment, the knowledge-rich tools are applied late in the coding cycle: the compiler provides feedback about the legality of the source only after the entire file has been produced, and the make facility uses dependency information only to manage compilation between files.

The earlier a problem is detected, the easier it is to correct. Newer software development environments often try to move the language-specific knowledge earlier into the coding cycle, and to use the information collected by such tools throughout the cycle in an integrated fashion. The environment then has knowledge about the objects it manipulates and their current state; it can respond interactively to errors and anomalies, and it can respond to queries about the objects' state [Medina-Mora and Feiler]. Lisp programming has long benefited from such language-specific environments as Interlisp [Teitelman and Masinter]. The development of language-oriented tools is an active area of research [Campbell and Kirslis], [Donzeau-Gouge, Huet, Kahn, and Lang], [Habermann], [Reiss], [Teitelbaum and Reps]; the programming environment to be provided for a language is now often a consideration in language design [Goldberg] [Teitelman].

The syntax-directed editor is an example of the application of language-specific knowledge early in the coding cycle. Such an editor is knowledgeable about the syntax of a particular language or languages. It ensures that the code entered is correct while the programmer enters it, providing immediate feedback about syntactic (and possibly semantic) errors and misuses. The editor may also provide the programmer with access to its knowledge about the language and about the source being edited--for instance, allowing the programmer to query about the followset of a particular token

[Campbell and Kirslis], or about the attributes of a defined identifier or scope [Reiss], [Teitelman].

A language-oriented editor must perform syntactic analysis, the first phase of traditional compilation. Usually the editor maintains the source file in structured form, as a syntax tree, rather than as linear text [Donzeau-Gouge, Huet, Kahn, and Lang], [Medina-Mora and Feiler], [Teitelbaum and Reps]. When a structured editor is used for program creation, the use of a standard compiler entails the unparsing of the source file, followed by redundant syntactic analysis.

Further, just as the programmer can benefit from the editor's feedback, the compiler can benefit from knowledge of which sections of a source file have been modified through editing. Such information can enable the compiler to recompile only the affected routines within a file, providing a separate-compilation-like facility for languages which do not support separate compilation (or support it only grudgingly).

1.2 Motivation

The SAGA project is investigating formal and practical aspects of computer support for the software lifecycle [Campbell and Kirslis]. Within the SAGA environment, epos is the language-oriented editor. The programmer enters code as with a standard text editor, but can manipulate syntactic entities as well as textual entities; epos incrementally parses and error-checks the code as it is entered.

Epos up to now has not had a semantic-evaluation component; it has only

checked syntactic constraints. Also, the editor maintains SAGA files as parse trees, rather than as text. Thus, compiling a SAGA file with a standard compiler entails unparsing followed by redundant syntax analysis.

SAGA Make [Badger] was originally designed for Pascal 6000 on the Cyber; that system supports the compilation of nested routines without compiling the routines which enclose them. Since Berkeley Unix's Pascal compiler pc does not support this, much of SAGA Make's functionality was lost when the SAGA system became Unix-oriented.

In environments which include syntax-directed editors, it is thus most efficient for compilers to leave the task of syntax analysis to the editor; such a compiler would generate code from the parse trees with which the editor works [Medina-Mora and Feiler]. SAGA Make demonstrates that the editor can be recording a modifications-trace as the programmer is modifying a pre-existing file; when such information is available, compilation is most efficient if it only involves the routines which were affected by the re-edit.

The system described here, pcg, is such a compilation facility for Pascal under SAGA. Pcg's symbol table component is a semantic-evaluation component added to epos; its code-generation phase is driven by the SAGA Make facility, and generates intermediate code directly from a traversal of the parse trees used by the SAGA editor. Use of Make enables it to recompile intermediate code incrementally upon re-edits of the Pascal source; this allows the programmer to keep a Pascal program in one unit, as Pascal encourages, but still have the efficiency of separate compilation.

A goal for tools in the SAGA system is that they form standard components which can be composed to form new tools. Pcg demonstrates the composition of SAGA tools to produce a new facility. Besides making use of information

generated by epos and Make, pcg uses the SAGA symbol table manager to store and organize the semantic attributes it collects, and to pass this information between phases.

Because it makes full use of the parse information collected by epos, Pcg eliminates the redundant syntactic analysis in compilations generated by the SAGA Make facility. Pcg is also a step towards making full use of the semantic information in the SAGA environment. Its symbol table component serves as a prototype interactive semantic component for the editor. It provides interactive response to semantic errors, and an ability to query the symbol table about the attributes of identifiers.

1.3 Previous Work

Above we noted the traditional role of compilation in programming environments; other divisions of labor between editor and translator are possible.

The classic alternative is the interpreter-based system which is standard for Lisp. Source code is maintained internally in linked-list form, which can be directly executed by the interpreter. The system routine which parses user input thus produces a representation which is simultaneously the internal representation of the source and its executable representation. Runtime access to the source's representation supports sophisticated debugging facilities. Use of a run-time symbol table enables the programmer to replace routines at will. Because compiled routines are likewise managed by the interpreter, and communicate with other routines via the symbol table, they

too may be replaced freely; but they lose most of the benefits of the debugging facilities. Interlisp [Teitelman and Masinter] is an advanced example of such a system.

MENTOR [Donzeau-Gouge, Huet, Kahn, and Lang], [Donzeau-Gouge, Lang, and Melese] also maintains program source in structured form; code for various languages is maintained as abstract syntax trees. General tree-manipulation tools are provided, and may be composed into procedures for manipulating particular languages. Editing is tree-oriented. MENTOR provides a variety of sophisticated interpreters to evaluate and transform the abstract syntax trees which represent programs. Some perform semantic checking. Compilation is performed with standard compilers, after unparsing the source into text form.

In the Cornell Program Synthesizer [Teitelbaum and Reps], source files are maintained as abstract syntax trees with associated symbol-tables, and an interpreter is provided which can directly execute these trees. Thus, although a compiler-oriented languages is used, compilation does not occur. The interpreter returns to the editor upon encountering a discontinuity in an incomplete tree, so a partial program can be run up to that point; this allows editing and testing to be highly interleaved. The standard Synthesizer is an educational rather than a development tool, and does not support compilation to machine code, nor separate compilation.

PECAN [Reiss] attempts to provide the user with multiple views of a program, including its syntax, semantics, and run-time behavior. Its compiler is oriented toward giving the user access to the semantics of programs. The user may query about the symbol table associated with a particular scope, including identifiers and their attributes; the compiler also supports the display of the expression tree representation of a given expression. PECAN's

design includes an interpreter, which will execute the internal form of programs.

Cedar [Teitelman] is a compiler-oriented language whose environment attempts to be interactive and experimental like interpretive environments. To this end, it provides both a compiler and an interpreter, which can interpret the full range of expressions of the language. Cedar's interpreter allows its user to query about the type of expressions, and evaluate type-valued expressions. The system keeps track of which files need to be recompiled, though dependency-analysis is not performed.

The Incremental Programming Environment [Medina-Mora and Feiler], under the Gandalf project [Habermann], is the system which most closely resembles pcg. It tries to provide the facilities and flexibility of interpreter-based systems entirely via compilation technology, and is oriented toward the production of long-lived programs. IPE generates machine code from the syntax trees which its syntax-directed editor produces, and performs incremental recompilation on the procedural level. Rather than generating a new executable object via a standard link-editor, as pcg does, IPE provides an incremental linker which can replace the machine-code version of a changed procedure within the executable object; it recompiles procedures in the background, rather than upon user request, as pcg does. Unlike pcg, it includes a debugger which is integrated with the rest of the system.

1.4 Overview

The design and implementation of pcg is described here. Chapter 2 details the overall structure of the major components of the system, and their design goals. Chapter 3 describes the implementation of pcg's first phase, which maintains the symbol table. In chapter 4 we look at the implementation of the second phase, which performs incremental recompilation. Chapter 5 summarizes what was accomplished, and points up shortcomings and directions for further research. Appendix A details the differences between pcg's Pascal and ANSI Standard Pascal, and between pcg's Pascal and Berkeley Pascal. Appendix B is a Unix manual page for the pcg incremental recompiler.

Chapter 2

DESIGN

Here we look at the design goals which pcg addresses, with particular attention to how it is designed to interact with the other tools in the SAGA system.

2.1 Overall Structure

Pcg decomposes into two phases which must be applied in sequence. In the semantic processing phase, pcg's symbol table component generates or updates the symbol table, given the program source in the form of a SAGA parse tree. In the compilation phase, the incremental recompilation driver of pcg takes the parse tree and symbol table, and compiles the program. The incremental recompilation driver relies on the code generator for the actual generation of intermediate code, which is transformed into machine code by the latter phases of the Berkeley Pascal compiler.

The symbol table component has two configurations. The editor-resident configuration constructs a symbol table concurrently with the editing of program source, and so can provide interactive feedback to the editor's user. Normally, the editor-resident symbol table component is invisible to the

user. If the user makes a semantic error, the symbol table component opens a window to emit an error message; also, the user can request information about the objects in the symbol table. The symbol table component can also be configured as a standalone program, which traverses a static parse tree to construct or update the symbol table for that program. This configuration is meant to be called by other SAGA tools.

When a syntactically-correct parse tree and semantically-correct symbol table are available, compilation can occur. This phase of pcg is invoked just as a standard compiler would be. The incremental recompilation driver controls the compilation process, using the modifications-trace generated by SAGA Make to determine which routines must be recompiled. For the routines which have been modified, or newly created, the driver calls the code generator, to generate intermediate code. The driver merges the new code with the unchanged code from previous compilations, and invokes the latter phases of the Berkeley Pascal compiler to complete compilation.

Figure 1 shows the interaction between the SAGA Pascal editor and pcg; the editor-resident symbol table component is displayed. The pcg system is separated into self-contained modules with well-defined interfaces, so that the modification or replacement of one component will not disrupt the functionality of the others.

Although SAGA syntax-directed editors have been generated for several languages, the Pascal editor is the base editor. Thus, pcg compiles Pascal. The language it accepts is currently a Pascal subset, which soon will be extended to full Pascal; see Appendix A. The particular Pascal dialect is Berkeley Pascal [Joy, Graham, and Haley].

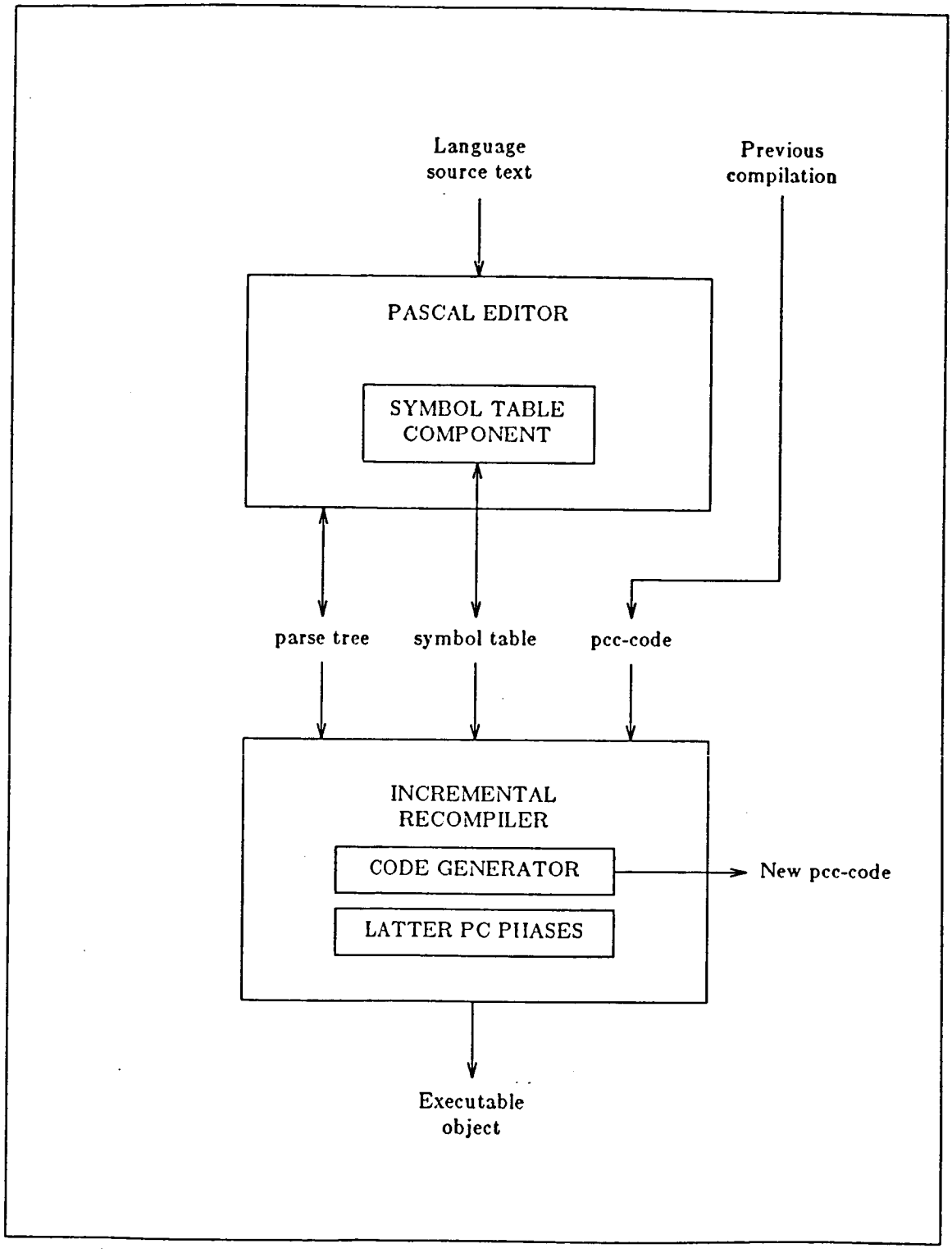


Figure 1. The SAGA Pascal editor and pcg.

Below we look at the design goals for the semantic phase's symbol table component, and the compilation phase's incremental recompilation driver and code generator.

2.2 Semantic Processing Phase

The symbol table component has two configurations, and serves as a prototype semantic component for the SAGA system. It had several design goals.

First, a goal for SAGA tools in general is that they form standardized, reusable modules which interact through well-defined interfaces [Campbell and Kirslis]. Therefore, the symbol table component tries to make as few assumptions as possible about epos and the internals of the parse tree files. To this end, pcg's symbol table component uses only the standard node-access interface to obtain parse-tree information; to communicate with the editor proper and with the programmer, it uses only the standard semantic-evaluation interface. Though dependence on the structure of the Pascal grammar is unavoidable, the symbol table component only assumes that the parse tree is well-structured with respect to that grammar, and that the tree's abstract internal relationships will not change without explicit editing actions. The symbol table component does not, for instance, store the internal node-indices of identifiers whose attributes it records, and thus the SAGA tree compactor could be run on a parse tree without invalidating the associated symbol table.

A second goal for the symbol table component is an ability to be

configured as a semantic evaluator, resident in the editor, or as a separate non-interactive process which performs semantic checking and symbol-table construction. Semantic evaluation can degrade the performance of syntax-directed editors [Medina-Mora and Feiler], and so the availability of a standalone configuration adds flexibility which may be needed when system resources are strained. In this configuration, the symbol table component resembles the semantic processing of a more traditional batch compiler.

A third quality sought in the symbol table component is the ability to collect information for two related but different tasks. 1) As the symbol-table constructor for the pcg compilation system, the symbol table component must collect the information needed for compilation. 2) Like a standard compiler, it also must be able to provide diagnostics about semantic errors and anomalies; additionally, to make use of the unique interactive capabilities of an editor-resident evaluator, the in-editor version can respond to user queries about the attributes of identifiers.

Fourth, in its role as a prototype semantic evaluator for epos, the symbol table component of pcg provides some support to incremental modification of the source program. Thus, when the user modifies the parse tree by re-editing, the in-editor symbol table component responds with consistent updates to (or deletions of) symbol table entries.

Finally, the symbol table component uses the SAGA Symbol Table Manager [Richards] for storing, organizing, and retrieving the attributes it detected. This is the first major exercising of the symbol table manager, which was designed as a general facility for software development environments in which multiple tools would have to exchange semantic information.

2.3 Compilation Phase

The compilation phase is managed by the incremental recompilation driver; the actual generation of intermediate code is performed by the code generator. We first look at the design decisions for the incremental recompiler as a whole.

The first design decision for the incremental recompiler was that the replacement unit for incremental updating would be the procedure. Interpreter-based incremental systems can update their executable code on the expression level [Teitelbaum and Reps]; interpreted code need not deal with the peculiarities of hardware, and can be designed to reflect the structure of the source language. By contrast, compiled code often bears only implicit structural similarity to the original source. Because the procedure or function is a self-contained unit with a well-defined interface, recompilation on the procedural level is a reasonable implementation for incremental recompilation [Medina-Mora and Feiler].

The next design decision which determined the structure of the incremental recompiler was that it would generate intermediate code, and use a standard machine-code generating second pass to complete compilation. A code generator was developed, which generates intermediate code from a parse tree and symbol table. The code produced is binary "portable C compiler" intermediate code [Kessler], hereafter called (with some inaccuracy) 'pcc-code'. This intermediate representation is a binary, packed version of the original portable C compiler intermediate code. It is largely

machine-independent. The Berkeley Pascal Compiler pc, and FORTRAN compiler f77, use pcc-code as their interface to the common machine-specific backend, which generates machine code. Use of this interface enhances the portability of pcg among Unix systems, with the other SAGA tools.

2.3.1 Incremental Recompilation Driver

When a user invokes pcg, the component of the pcg system that responds is the incremental recompilation driver. This component is the top level for the compilation phase of pcg. Given a SAGA Pascal file, it does what is necessary to ensure that its executable object is up-to-date with respect to its source.

The first design decision for the driver was that it would use SAGA Make's modifications-trace as a guide to generating new intermediate code. SAGA Make [Badger] was designed to be a largely language-independent facility in two phases. Its first phase, resident in the editor, keeps track of which routines are modified, or have their environments modified, such that they must be recompiled. Make's second phase used this modifications-trace to build a shell script which would recompile the program, and then it executed that script; this phase suffered from Berkeley Pascal's lack of facilities for compiling nested routines.

A goal met by virtue of using Make is that the code generator need only recompile the minimal number of routines necessary for updating the pcc-code and regenerating the object [Badger]. Pcg therefore preserves the pcc-code file which resulted from the most recent compilation, so that unmodified routines can be reused. Alternately, the incremental recompiler can be ordered to discard the old pcc-code and regenerate the entire program from

scratch.

A third property sought in the design of the incremental recompiler is that its user interface appear as similar to that of a standard compiler as possible. Pcg can, therefore, be invoked from within Unix make scripts just as can pc. Similarly, pcg can easily interface with configuration management schemes which make use of standard compilers [Kirslis, Terwilliger, and Campbell], [Estublier, Ghoul, and Krakowiak].

This decision implies that pcg does not perform compilations in the background during editing, as does IPE [Medina-Mora and Feiler]. However, if such a system were desired, epos's capability of spawning filter processes could straightforwardly implement it.

The incremental recompilation driver tries to behave reasonably if given a SAGA file for which no symbol table, or no modifications-trace, exists, invoking the standalone symbol table component to build a symbol table if necessary.

2.3.2 Code Generation

The code-generator is modeled on the first pass of the Berkeley Pascal compiler. It produces pcc-code, which the driver then provides to the later phases of pc.

As a simplifying assumption, the code generator follows pc's internal logic and algorithms wherever possible. The Berkeley Pascal compiler has proven itself as a tool for software development; most of the SAGA system, including most of pcg itself, is compiled with pc. Pc provides a reasonable separate compilation facility, and the ability to call routines written in other portable C compiler - based languages, including Unix system calls [Joy,

Graham, and Haley].

This design decision allows pcg to use pc's latter phases unchanged. Basing the code generator on the Berkeley compiler also enables a simple test of its output: if the pcc-code that it generates differs in structure from that generated by pc for a given Pascal program, then something untoward is going on.

For the sake of modularity, the code-generator's job was limited to producing pcc-code, given a parse tree, a symbol table, and a node which is the root of a subtree for a routine. Managing the further phases of compilation is left to the incremental recompiler.

The next chapters provides an overview of the implementation which tries to meet these criteria.

Chapter 3

SEMANTIC PHASE IMPLEMENTATION

These two chapters examine significant implementation details of pcg. In looking at the issues in its implementation, we pay special attention to pcg's interaction with the other tools in the SAGA system, and with the Berkeley Pascal compiler.

In this chapter we will look at the implementation of the symbol table component of pcg, which performs the semantic phase of pcg's processing; in the next, we will look at the compilation phase.

3.1 Semantic Processing

The problem of semantic analysis of programs is nontrivial. The syntactic task of parsing has been simplified by the development of the context free grammar formalism [Aho and Ullman], to the extent that automated tools such as YACC [Johnson2] and Mystro [Noonan and Collins] can construct parsers from a formal description of a grammar. But no fully satisfactory formalism for semantics has been developed, although attribute-grammar based systems for automated semantic analysis and compilation are an active research area [Paulson], [Ganapathi and Fischer], [Reps].

[Reps] distinguishes between imperative and declarative semantic evaluators. The former is procedurally specified, the latter uses a formal specification to enable the automatic generation of an evaluator. Imperative evaluators must specify both semantic actions, which are to be performed upon the insertion of program text, and semantic retractions, which update the symbol table when a deletion occurs.

The declarative method attempts to avoid the need for retractions, by eschewing the use of a global symbol table whose state must be kept consistent with the state of the syntax tree. Rather, it stores semantic information locally, throughout an attributed tree. It is unclear whether such localized context is sufficient in general [Johnson and Fischer]. An attribute-grammar based evaluator, combining both declarative and imperative aspects, is under development for the SAGA environment [Beshers and Campbell]. In the meantime, the symbol table component of pcg provides the pcg system with an ad-hoc, imperative mechanism for collecting semantic attributes and error-checking SAGA Pascal source.

3.2 The User Interface

To the user of epos, the symbol table component of pcg is merely another feature in the editor. The editor proper reports when the user enters syntactically-incorrect text, and highlights the unparseable portion of the program. Similarly, the symbol table component opens a window and emits an error message when the user enters a semantically-incorrect declaration or statement; the offending string within the program is highlighted. If the

user modifies a declaration in such a way that previously-entered text which depends on that declaration is now incorrect, the error is reported and the now-incorrect strings highlighted. The attempt to re-declare an identifier, within the same block as a previous declaration of that identifier, causes the generation of an error message, the highlighting of the offending identifier, and the disregarding of the new declaration. If a new identifier is entered with a semantically-malformed declaration, the identifier is entered into the symbol table, but its attributes note that it is misdeclared. Upon correction of an error, the corrected code is displayed in the normal font again.

The editor-resident symbol table component also provides the user with the ability to query the symbol table about the attributes of currently-defined identifiers, including both standard and user-defined types, variables, and routines. A similar facility is provided in PECAN [Reiss] and Cedar [Teitelman]. This facility is particularly useful in a separate compilation environment; for instance, one can check the number and types of the parameters of an imported routine, before entering a call to that routine. It is also useful when the symbol table component informs the user that a symbol has been misused; the symbol's attributes can be inspected, to determine how to correct the mistake. Normally, the search for an identifier starts in the current block and proceeds outwards until a definition is found. It is also possible to enquire about symbols defined within contexts which are nested within the user's current context; one prefixes the identifier with a path of context names separated by dots. For instance, to enquire about the field 'i' within the record 'rec', declared within the nested function 'ftn', one enquires about 'ftn.rec.i'.

The standalone symbol table component is oriented toward use as a tool by

other tools, unlike the editor-resident configuration of the symbol table component. The standalone configuration is a self-contained program that takes one argument, a SAGA file name. It loads an existing symbol table, if present, and then traverses the parse tree, to produce an updated symbol table. Nodes generating semantic errors are marked in the parse tree, and the error messages written to standard output.

3.3 Overall Structure

The task of semantic analysis is significantly complicated by a need to support incremental modifications of the program source. Existing declarations can be modified or deleted, necessitating the change or removal of symbol table entries; such changes can correct or invalidate other entries which reference the objects declared. Existing executable statements are also subject to modification or deletion, and must be re-checked for legality. The user of a syntax-directed editor can enter syntactically-incorrect or incomplete code, but the symbol table must not thereby be left in an inconsistent state.

Pcg's symbol table component is an imperative evaluator, since the semantic analysis is specified procedurally; it binds action and retraction procedures to grammar productions. When the editor reduces by such a production, or when the standalone symbol table component encounters such a production during tree traversal, then the associated procedure is invoked. The procedure traverses the affected subtree to gather needed information, and then it updates the symbol table.

Below we explore the linkages between the grammar of Pascal and the symbol table component; this provides background for understanding the use of actions and retractions. Next we look at the symbol table component's use of the SAGA symbol table manager; in this context, the use of action and retraction routines is described.

3.4 The Symbol Table Component and the Pascal Grammar

To support incremental evaluation, pcg's symbol table component must respond appropriately to modifications in program text. To this end, it distinguishes three special subsets of the production rules in the LALR(1) Pascal grammar used by the current Mystro-based SAGA editor [Aho and Ullman], [Noonan and Collins]. These subsets are the action productions, the checkable productions, and the user productions.

3.4.1 Action productions

Certain productions are distinguished as being 'action productions'. When an action production is encountered, an entry is installed into the symbol table. In general, an action production roots a subtree of least height such that the subtree contains all the information needed to determine the attributes of an identifier. By delaying until all information needed is present, the symbol table component does not need to maintain external data structures containing partial attributes, which would have to be handled specially if user input were interrupted or discovered to be syntactically malformed. On the other hand, by not delaying until a reduction to a

higher-level nonterminal is performed, the symbol table component can respond most immediately to erroneous input.

3.4.2 Checkable productions

A single production lies in the set of 'checkable productions'. This is the production whose left hand side is <statement>. Within a statement, expressions must be typechecked, the use of expressions must be checked for legality, and references to declared entities must be recorded. Such actions are performed when the symbol table component encounters a reduction to <statement>.

3.4.3 User productions

The third subset of Pascal grammar rules is the set of 'user productions'. These are the productions which contain user-supplied terminals; reduction by such a grammar rule, during the non-reparsing first phase of the parse, indicates that a tree modification has occurred, which should be analyzed. The user productions are significant in the editor-resident semantic phase. The epos parser is incremental, and attempts to reparse the minimal amount needed to fit changes into the parse tree [Ghezzi and Mandrioli]; where possible, it shifts entire subtrees, rather than their frontiers. It is thus possible that a user modification can be accommodated into the tree, without the reparse propagating up to the action or checkable production which is its ancestor. If a reduction by a user production was not eventually followed with a reduction by an action or checkable production, the symbol table component detects the need to climb to

that ancestor and re-evaluate the subtree it roots.

3.5 The Symbol Table Component and the Symbol Table Manager

Much of the work of the symbol table component is simplified by its use of the SAGA symbol table manager.

3.5.1 Attributes

Use of the symbol table manager is organized around the attributes which one sets up for the given application. Symbol table manager primitives are used to record symbol definitions and symbol references; (attribute, value) pairs can be attached to such entries. The symbol table manager's user must specify what type the value of an attribute may take on.

Attributes are identified by strings stored in the symbol table's strings section; referring to a particular attribute is accomplished by a reference to that string's internal identifying tag. Thus, for every attribute one defines, one must maintain a variable containing that tag, to enable one to refer to the attribute. This is an impetus toward defining record-valued attributes; such an attribute-complex can hold all the values associated with a given class of symbol.

By making the user-defined attribute type a variant record, it can be used for several attributes. The symbol table component uses the user-defined attribute type for four such 'compound attributes'; the two most important are called NameAttributes and TypeDefAttributes.

The symbol table component's use of these attributes is straightforward.

Consider an example of an action routine. When the symbol table component encounters the production

<var_decl_list> ::= <variable_list> : <type>

it invokes an action routine to inspect the <type> subtree. If it is an actual type definition, then the subtree is traversed and the attributes of the type collected. For example, if the subtree defines a subrange type, the host type and endpoints are recorded. The routine returns an anonymous type-definition symbol, which has one attribute containing the description of that type. Alternately, the <type> subtree may not be a new type definition, but an identifier: a reference to a previously-declared type. The symbol table entry bound to that identifier is retrieved, and its NameAttributes inspected to determine which anonymous type-definition symbol it names.

In either case, once the <type> subtree has been handled, another action routine traverses the list of variables. For each, it inserts a non-anonymous symbol, to be known by the identifier indicated; the new symbol's NameAttributes specify that it names a variable, whose type is that previously-obtained type-definition symbol.

The incremental parser within epos must sometimes reparse previously-analyzed code, to analyze new material inserted into that code. The possibility arises that an action routine would be called a second time, causing a spurious "identifier previously declared" error. An addition has been made to epos's semantic interface which notifies the symbol table component when a reparse moves into previously-parsed text; action routines are not called for such reductions.

3.5.2 Contexts

With the symbol table manager, when one inserts a symbol definition or symbol reference, one indicates the 'context' in which to place it. The main program, each procedure, each function, and each record type, has an associated context. Identifiers declared within blocks or records are stored within their contexts. To retrieve a symbol, given an identifier, one specifies a context in which to search; contexts can be nested, and searches proceed from an inner context outward. This makes the implementation of Pascal's block-structured scoping rules trivial.

More complex context interactions are generated by the use of grafted contexts. Thus, for example, when pcg enters the scope of a Pascal 'with' statement, it grafts a temporary context onto the current block's context. When a variable is encountered, the search for its definition is first performed in the context of the indicated record, seeking the identifier as a field; then in the current block, seeking it as a variable; and then outwards in any outer blocks. Variables and fields can therefore be handled by the same code; use of the symbol table manager promotes the orthogonal manipulation of symbols.

3.5.3 Symbol References

Besides symbol definitions, the symbol table manager also supports the recording of symbol references. If a symbol is referred to in a given context, a reference entry can be made, and attributes given to it; the symbol definition can be recovered from the symbol reference, and any recorded

references can be recovered from the definition. Further, if a definition is deleted, but there exists a definition of that identifier in an outer block, any references made to the deleted symbol becomes attached to the now-visible outer definition.

This automatic action of the symbol table manager is very useful in dealing with deletions in an incremental environment. In the standard Cornell Program Synthesizer, for instance, the deletion of a declaration invalidates the entire symbol table, and necessitates re-traversing the entire parse tree to build a new one [Teitelbaum and Reps].[2] In pcg's symbol table component, outer blocks are not invalidated, since the deleted symbol was invisible there, and any nested blocks which do not refer to the deleted symbol need not be re-evaluated.

3.5.4 Retractions, Attributes, and References

We saw above that the action routines are grammar-driven. In contrast, the retraction routines are driven more by the structure of the attribute-records. The top-level retraction routine traverses the subtree given to it, seeking definitions of identifiers. On encountering such a definition, the identifier's attributes are retrieved from the symbol table, and further actions are based on those attributes. Consider the variable declaration described above. The variable's symbol table entry must be deleted. The type recorded for it is also inspected. If its attributes indicate that no identifier was bound to it, then the type definition entry is

---

2. This is handled more economically in Synthesizer-Generator based systems, which use attributed trees rather than a standard symbol table [Reps].

deleted. Otherwise, the entry which records that the variable referenced the type is deleted.

If references to the deleted entry existed, then the contexts which made those references are noted. Upon completion of the retraction, those contexts are re-evaluated, to ensure their validity. Re-evaluation consists simply of the retraction of entries defined in the routine's subtree, followed by a new tree traversal to re-install these entries and re-inspect the routine's executable statements.

3.5.5 Other Features and Limitations

Each symbol table primitive returns an error code. This provides considerable consistency-checking to the symbol table component; if an internal error occurs, then at some point a symbol-table primitive will be unable to complete its task, and an error will be reported.

Limitations of the prototype symbol table manager also affect the symbol table component. No provision is made for anonymous symbols, nor for lists of symbols; the symbol table component must simulate these features.

The symbol table manager is oriented toward the support of separate compilation, by allowing multiple symbol tables to be open simultaneously; however, the support presently provided is limited by the requirement that each such table have a unique permanent identifier. This prevents the re-use of standard modules, if the permanent-ids assigned to them clash with the identifiers of other modules already in use. A new version of the symbol table manager has been proposed; this new version will provide a virtual naming scheme for multiple open tables. Because this version is not currently available, pcg does not yet support separate compilation.

In the Berkeley Pascal model of separate compilation, included header files contain declarations of external entities; these are considered to be global, that is, declared at the level of the main program context. Although the incremental recompiler can compile separate code modules, the limitation mentioned above makes it is currently impossible for references to be made across modules. Enabling separate compilation in pcg should not be difficult when the new facility becomes available.

The next chapter is an overview of the implementation of pcg's next phase, the incremental recompilation phase.

Chapter 4

COMPILATION PHASE IMPLEMENTATION

Here we examine some of the issues involved in the implementation of the compilation phase of pcg. The major work of compilation is performed by the code generator, which generates pcc-code from a SAGA parse tree and a symbol table. Incremental recompilation is achieved by the incremental recompilation driver, which calls the code generator as needed to generate new code for modified routines. First, we look at the code generator.

4.1 Code Generation

The code-generator is very similar to the pc0 phase of the Berkeley Pascal compiler. It is essentially a translation into Pascal of the relevant parts of that program; instead of pc's namelist and parse tree, the SAGA symbol table and parse tree are its input. As output, it produces the same sort of Portable C compiler intermediate code as pc0 produces.

To examine the code generation component of pcg, we will first look at pcc-code itself, and its use in representing Pascal programs. Next we view the overall structure of the Berkeley Pascal compiler, and the system it implements. Then we will be ready to examine the general structure of the pcg

code generator.

4.1.1 Pcc-code

The structure and content of pcc-code is described in [Kessler]; the philosophy and organization of the Portable C compiler is detailed in [Johnson1].

Pcc-code is a postorder linearization of the binary expression trees, and flow-of-control operators, produced by the Portable C compiler to represent C code. It makes explicit the content of the original C program, and decomposes it into simpler structures. For instance, in pcc-code, all operators and operands are explicitly typed, and needed conversion operators inserted. Also, C's structured statements are converted into simple tests and jumps.

Much of pcc-code is machine independent. The first pass is required to handle certain machine dependent constructs, such as routine prologues and epilogues, the code for switch (that is, case) statements, and initializations. This is done by emitting assembly code which will be passed unchanged through the next pass, which generates assembler from pcc-code.

Since pcc-code was designed to represent C, there is some mismatch to be dealt with in representing Pascal code. To represent Pascal expressions, C's wealth of operators are more than sufficient; many pcc-code operators are never used by pc0. On the other hand, Pascal's rich type structure sometimes requires simulation; several Pascal types (for instance, sets) are by default represented as C structures, and operations on these types are implemented by library functions. (The overhead thus incurred is obviated somewhat by the pc2 phase of the Berkeley compiler, described below.) C's structure type is convenient for such use because it is a structured type which may be the

target of assignment, may be passed to functions, and may be returned by functions. (But C's support for these operations on structures causes some complication in pcc-code; pcc-code must assume, for instance, that the value of a structure-valued expression is actually a pointer to a structure, rather than the structure itself.)

4.1.2 Structure of the Berkeley Pascal Compiler

The Berkeley Pascal compiler is a five-pass compiler. The first pass, pc0, does syntax analysis, semantic checking, and generation of pcc-code. The second pass, pc1, is actually the f1 pass of the f77 FORTRAN compiler; this is the pass derived from the second pass of the Portable C compiler, which takes binary pcc-code as input and produces assembler as output. The resulting assembly language is the input to pc2, the inline expander. This filter passes most of the assembler unchanged; calls on frequently-used system functions are expanded in place into the assembly code which implements them.

Pc2's output is given to the Unix assembler as, which produces unlinked binary. The pc3 phase examines the symbol tables of binaries produced in this way, prior to linking; it does several checks on the use of globally-visible routines and variables, to enforce the rules of separate compilation in Berkeley Pascal. Finally, the binary is link-edited via Unix's ld, to produce an executable object.

Because the pcg code generator produces pcc-code such as the pc0 phase would produce, pcg can run the latter four phases of pc unchanged. Thus, pc0 is the pass most of interest here.

Pc0 is driven by its YACC-based parser [Johnson2]. The parser constructs the parse tree such that the structure of a subtree can be determined by

examining its first node. As the parser recognizes declarations, routines are invoked to make entries in pc0's namelist (symbol table). The structure of namelist entries is a bit baroque, consisting of many overloaded fields, rather than a variant record structure such as is encouraged by the SAGA symbol table manager. Whenever the parser recognizes a complete procedure, function, or program, a function is invoked which traverses the resulting subtree simultaneously to check semantics and to generate pcc-code.

The runtime system created by the Berkeley compiler is essentially that of the Berkeley Pascal interpreter px, as described in [Joy and McKusick]. Px defines many system functions to implement both Pascal operators and built-in routines, such as the input and output procedures. This simplifies the use of this run-time system with C-oriented pcc-code; where pcc-code is deficient, the appropriate library function can be used. The px runtime system is almost purely stack oriented. The objects operated on are assumed to be on the stack, or else in the heap area, and are operated on by the interpreter's Pascal-oriented operators. In contrast, the pc system's use of pcc-code enables it to make use of the abilities of the f1 code generator, which generates assembly code targeted for the actual hardware, and attempts to place operands in registers as much as possible. Pc uses the stack for activation records, structured objects, parameter-passing, and extra temporaries. A display is maintained for referencing nonlocal variables from nested routines.

4.1.3 Structure of the Pcg Code Generator

The interface to pcg's code generator is simple. It takes a node, a context, and a job-specification; the node must be the root of a procedure,

function, or program subtree, and the context must be the symbol-table context associated with that routine. Based on the job-specification, the code generator either generates code for the indicated routine, or else performs semantic checks on the statements within the routine.

For ease of interfacing with the other SAGA tools, particularly the symbol table manager, the code generator is implemented in Pascal. The low-level routines which actually produce the binary pcc-code are written in C, as are a set of routine which are used for bit-level operations which are occasionally necessary.

[Medina-Mora and Feiler] note that an advantage of compiler-based environments over those which are interpreter-based is the ability to produce code for a target machine which is different from the host on which the environment is running. The current implementation of the pcg code generator is targeted for the VAX[3]. The pc sources can be configured to generate code for the VAX or for the MC68000, and this capability has been provided in pcg, although the 68000-oriented code-generator has not been tested.

The pcg code generator routines can be partitioned into four sets: those which interface with the symbol table; those which actually walk the parse tree and generate intermediate code; those support routines which implement the machine-dependent aspects of code generation; and those support routines which implement the aspects of code generation dependent on the Pascal runtime system.

---

3. Vax is a trademark of Digital Equipment Corporation.

4.1.3.1 Symbol table interface.

To prevent too tight a coupling between the symbol table component and the code generator component, all symbol table accesses are isolated into a set of routines which are invoked to query the symbol table, and to change the context. Thus, for example, predicates are provided to indicate the attributes of types and variables; the isintegral predicate returns true if its argument is type integer, or a user-defined type which is a subrange of integer. Similarly, graftrecordcontext grafts a temporary context onto the current context, to implement the scoping effect of a Pascal 'with' statement or field selector. This modularity should ease the transition to the attribute-grammar based evaluator planned for the SAGA system.

4.1.3.2 Code-producing routines.

The pcc-code producing routines walk the parse tree to emit C code. Because they must walk the tree, they are very dependent on the structure of the Pascal grammar; for instance, the structure expected in a subtree is determined by checking its production number. This tight coupling is slightly alleviated by the usage of symbollic names (Pascal constants) for the rule-numbers in the grammar; however, there is no way to eliminate the dependence on the internal structure of the productions.

The code-producing routines mirror the Algol-family structure of Pascal and C. The top-level routine generates code for a procedure, function, or program; it handles program unit prologues and epilogues, and the emitting of symbol table directives which provide information to pc3 and the Unix

debugger. It invokes other routines to deal with the executable statements in the program unit's body.

For each Pascal statement, there is a procedure to traverse its subtree and emit code; these emit the flow-of-control operators. At the bottom level are the routines to generate code for l-values (locations) and r-values (expressions); these emit the pcc-code expression trees.

4.1.3.3 Machine-dependent aspects.

The third class of routines in the code generator are those which implement machine-dependent aspects of code generation. An example is the alignment module, which is used by by the symbol table component to allocate offsets for variables; another is the temporaries module, which handles the allocation of temporary variables for the current block (placing them in registers when possible).

4.1.3.4 Runtime system routines.

The fourth group of routines are those which support the use of the Berkeley Pascal run-time system. A good example of this group is the sets module. Routines from this module have diverse duties relating to the Pascal set type, such as determining whether a set expression is a constant set, determining the type of a constant set, or emitting the proper Pascal-system function call to perform the indicated set operation.

4.2 Incremental Recompilation

The code generator component of pcg is controlled by the incremental recompilation driver. The driver for the pcg incremental recompiler is straightforward. When pcg is invoked to compile a SAGA file, the driver first checks that a symbol table file exists within the SAGA directory which implements the SAGA file; if no symbol table exists, the standalone symbol table component is invoked to generate one. Next, the incremental recompiler checks that a modifications-trace is available. If not, then it assumes that the entire file is to be recompiled. Alternately, the user may demand that the incremental recompiler ignore the modifications-trace, and recompile the entire file.

When a SAGA source file has been previously compiled with pcg, its SAGA directory will contain two additional file. One is the pcc-code which resulted from the last compilation. The other is a list of the routines present in that file; for each routine, the location of its last word of code, within the pcc-code file, is recorded.

The process of recompilation is a simultaneous post-order traversal of three tree of routines: the tree of routines represented by the parse tree, and the linearizations of that tree present in the two files described above. For each routine in the parse tree, if the modifications-trace indicates that the routine must be recompiled (or if the modifications-trace is unavailable), then the code-generator is invoked to generate new pcc-code from the routine's subtree and its context in the symbol table. The pointer into the file of old

pcc-code is advanced past the routine. If the modifications-trace indicates that the routine need not be recompiled, then its pcc-code from the old compilation is copied verabatim into the new file, advancing the pointer.

Pcg then invokes the later phases of the Berkeley Pascal compiler, with the new pcc-code as input, to complete the compilation. If the -c (separate compilation) option was specified, then the last two phases of pc are not run, and the result of the compilation is an unlinked object, just as with pc. If the separate compilation option was not invoked, then an executable object is produced. In either case, the process produces three other files: a new pcc-code file, a new routine-locations file, and a modifications trace which now indicates that all routines are up-to-date.

Chapter 5

CONCLUSION

Pcg demonstrates that a compiler in a language-oriented environment can make use of the information gathered by other tools to improve the efficiency of compilation. The parse trees produced by the epos syntax-directed editor are sufficient for compilation; an interactive semantic evaluator, implemented with the symbol table manager, can build a symbol table to enable compilation; and the modifications-trace collected by SAGA Make can be used to eliminate redundant compilations. The final result shows the usefulness of tools which share information to avoid duplication of effort.

Pcg demonstrates the composition of tools in the SAGA environment. The SAGA tools pcg uses had not previously all been required to cooperate simultaneously. Occasionally a tool did not correctly implement its interface, or the interfaces of two tools clashed so that they could not communicate with each other without difficulty. Though such real-world difficulties occurred, the tools were composed to generate a complex application.

5.1 Statistics for Example Programs

Pcg moves the task of symbol table construction, along with the task of syntactic analysis, from the translation phase of the coding cycle into the editing phase. Further, it attempts to improve the efficiency of compilation by incrementally compiling within files. We consider figures on time and space costs collected for two sample programs.

	declarations.p (147 lines)	pxref.p (389 lines)
epos without semantic evaluation	11.0 user seconds 3.0 system seconds	43.8 user seconds 6.7 system seconds
epos with semantic evaluation	23.2 user seconds 4.8 system seconds	61.3 user seconds 9.6 system seconds

Table 1. Times required for the editor to read and analyze two files.

Table 1 shows the time required for epos to read in and analyze two files: the first consists entirely of complex declarations; the second, Wirth's cross-reference program, is a more realistic mix of declarations and code. In the first case, the symbol table component makes the editor run approximately twice as slow. These worst-case figures may be misleading. In actual interactive editing, the time cost of semantic evaluation is spread out over many interactions; subjectively, the response time of the editor does not deteriorate significantly when semantics evaluation is included.

	declarations.p	pxref.p
standard text	3682	7955
executable object	15360	27648
pcc-code	4308	38592
parse tree	69644	262156

Table 2. Size in bytes of four representations of two files.

Table 2 shows that, although the pcc-code representation can be significantly larger than the straight text representation of a given program, it is not expensive compared to the current SAGA parse tree representation. Thus, preserving pcc-code files between compilations is a reasonable course.

	declarations.p	pxref.p
pc0	0.8 user seconds 0.8 system seconds	5.7 user seconds 1.1 system seconds
pc	4.0 user seconds 3.7 system seconds	31.6 user seconds 6.8 system seconds
code generator		
pcg		

Table 3. Compilation times for two files, in which one 20-line procedure was modified.

Does pcg improve the efficiency of compilations? Table 3 shows that pcg performs code generation faster than pc0, but, unfortunately, the first phase consumes only about a fifth of the time of a compilation. The latter four phases of compilation are shared by pc and pcg; pcg's incremental recompilation efforts are aimed at efficiently producing an intermediate code version of a file, which must then be given to the non-incremental pc backend to complete compilation.

Certain implementation problems remain. As a prototype, pcg is

insufficient for a true development environment. It will soon be extended to support full Pascal, but it must also support separate compilation if it is to be useful; this requires the resolution of the limitation in the symbol table manager previously mentioned.

5.2 Future Directions

Pcg suggests several directions for future work.

The most fundamental limitation of the pcg system is its dependence on a non-incremental machine-code generator. Any efficiency gained from incremental recompilation in the early phase is overshadowed by the time required to recompile the resulting code non-incrementally. A straightforward extension to pcg would be a facility for merging assembly-language rather than intermediate-code files; preserving assembly-language between compilations would make the first machine-dependent phase of compilation incremental. But recompilation should be incremental throughout all phases. A facility for merging binaries, such as existed in the original Cyber-based SAGA Make, or an incremental loader, such as in IPE, is required. Once such a facility is provided, pcg-style code generators can be developed for a variety of languages, and use the common backend.

If one is willing to sacrifice language-independence, then SAGA Make can be made more efficient, by using the symbol table manager's ability to record symbol references. Nested routines which do not reference a modified declaration in the outer environment need not be recompiled in response to that modification. Further, when the ability to use multiple symbol tables is

realized, it will be possible to record inter-file dependencies on the procedural level; this would make it possible, for instance, to avoid recompiling a file which references an unchanged interface even though the interface resides in a file where other interfaces were modified.

Pcg only deals with Pascal. The SAGA environment is meant to support several programming languages [Campbell and Kirslis]; SAGA editors exist for Pascal, C, Ada, and Backus' FP. Since pcc-code is also used to implement FORTRAN and C, the development of pcg-type compilers for these languages would be straightforward. As we have seen, the use of standard compilers which expect text input is inappropriate for an environment such as SAGA. But the hand-coded production of pcg-style code generators could be prohibitively costly in human time, for an environment which supports many languages. The addition of the attribute-grammar based semantic evaluator to SAGA will make the production of symbol table components far less ad-hoc. Since the symbol table component is a major part of a code generating system, producing such systems will become much less costly. The attribute-grammar specification for one language, which details the attributes needed to generate a given intermediate code from that language, could serve as the basis for developing specifications for other languages which will use that same intermediate code. Also promising is research on the automatic generation of compilers from attribute-grammar specifications of a language and an architecture [Ganapathi and Fischer], [Paulson]. It may be that such a formal specification can be used to generate an entire language-based environment, including editor, compiler, and debugger.

Pcg incrementally recompiles on the procedural level. Just as in IPE, a natural development would be the integration of a source-level debugger into

the editor/recompiler system, giving the ability to immediately run the actual machine code routines on sample input. This would enable rapid interleaving of program creation with program testing, as is possible in an interpreter-based environment. But by incrementally recompiling rather than interpreting, the true machine-code implementation would be the object of debugging, and the faster execution characteristic of non-interpreted code would be available.

Appendix A

PCG PASCAL, STANDARD PASCAL, AND BERKELEY PASCAL

A.1 Compliance with ANSI/IEEE 770 X3.97-1983 Standard Pascal

The SAGA pcg system complies with the requirements of ANSI/IEEE 770 X3.97-1983 with the following exceptions:

6.1.1. The case of letters making up identifiers and reserved words is significant. This follows the Unix convention.

6.1.3. Identifiers cannot be longer than 127 characters in length.

6.1.4. The directive #include may occur outside procedure-declarations and function-declarations.

6.1.5. Integers occupy the range minint..maxint, where minint = -2147483648, and maxint = 2147483647.

6.1.6. Labels may be longer than four digits in length; a warning is issued if such a label is declared.

6.1.8. If a comment begins with one type of delimiter and ends with another, a warning is issued. Nested comments are allcwed.

6.2.2.10. The required identifiers 'write' and 'writeln' have special

significance within the grammar, and should not be redeclared.

6.4.3.1. (The keyword packed has no effect.)

6.4.3.2. To be a string, an array of characters need not be packed, and the lower limit of its subscript need not be 1.

6.4.3.5. The predefined type 'text' is equivalent to 'file of char'.

6.8.3.5. The case statement is currently not implemented.

6.8.3.9. The for statement is currently not implemented.

A.2 Differences between Pcg Pascal and Berkeley Pascal

This section constitutes an addendum to Appendix A of the Berkeley Pascal User's Manual. See that manual for a full description of Berkeley Pascal.

A.1. Extensions to the language Pascal.

String Padding. Pcg Pascal pads constant strings with blanks as necessary, just as Berkeley Pascal does.

Octal constants, octal and hexadecimal write. Pcg does not support these Berkeley extensions.

Assert statement. The assert statement is not supported.

Enumerated type input-output. Pcg Pascal performs the nonstandard extension of enumerated type input-output just as does Berkeley Pascal.

Structure returning functions. Pcg Pascal allow functions to return records, sets, and arrays, just as Berkeley Pascal does.

A.1. Resolution of the undefined specifications.

File name - file variable associations. Pcg Pascal associates Pascal

file variables with named Unix files following the Berkeley conventions.

The files input and output. These are handled as in Berkeley Pascal.

Buffering. The buffering of 'output' is controlled by the b option, just as with Berkeley.

The character set. Just as in Berkeley, upper and lower case are distinct, and all keywords and required identifiers are expected to be all lower case. Use of ~, &, |, and # as synonyms for not, and, or, and ', are not supported.

Comments. Comments that start with one style of delimiter and end with another cause a warning message, as in Berkeley.

Option control. Options may be set in the pcg command line, in the standard Unix convention. Pcg Pascal does not support the control of options via flags in comments. See Appendix B for the options available.

Listings. No listings are produced. When errors are detected, their locations are indicted by setting a flag in the token causing the error; the token is thereby highlighted in epos's screen mode.

A.3. Restrictions and limitations.

Statements. Pcg Pascal does not currently support the following statements: goto, case, and for.

Files. The restriction that files cannot contain files is now part of the standard. As in Berkeley Pascal, files are also restricted from being members of dynamically-allocated structures.

Arrays, sets, and strings. The Berkeley restriction applies: arrays--including strings--and sets may have no more than 655355 elements; array and string subscripts are limited to the range -32768..32767.

Line and symbol length. Symbols are limited to 127 characters in

length.

Procedure and function nesting and program size. The arbitrary restriction of a maximum nesting depth of 20 is maintained in pcg. There is an unknown maximum program size; it is comfortable large.

Overflow. As Berkeley notes, the Vax does overflow checking in hardware.

A.4. Added types, operators, procedures, and functions

Additional predefined types. Alfa is predefined (and may be redeclared, of course). Intset is predefined to be set of 0..127.

Additional predefined operators. '<' and '>' may be used on sets to test for proper set inclusion, as in Berkeley Pascal.

Non-standard procedures. The following Berkeley non-standard procedures are supported by pcg: argv, flush, halt, remove, and the extended two-argument reset and rewrite. These are not supported: date, linelimit, message, null, stlimit, and time.

Non-standard functions. The following Berkeley non-standard functions are supported: argc, card, and expo. These are not supported: clock, random, seed, sysclock, and wallclock.

Appendix B

MANUAL PAGE FOR PCG

NAME

pcg - Pascal code generator

SYNOPSIS

pcg [option] filename...

DESCRIPTION

Pcg functions as a Pascal compiler in the SAGA Pascal environment. If given an argument SAGA file ending with .p, it will compile the file and load it into an executable file, called, by default, a.out.

Pcg currently does not support the following Pascal statements: case, goto, for.

Pcg compiles directly from the parse tree representation of the source file used by epos. Pcg expects the SAGA file (directory) to include a symbol table, generated by the epos-resident symbol table component of pcg; but in the absence of a symbol table, pcg will generate one. If the file was compiled previously with pcg, then a subsequent recompilation will reuse unchanged procedures from the previous compilation, for efficiency's sake.

Currently, pcg does not support separate compilation. When such support becomes available, it will be modeled on the example of Berkeley pc; see pc(1).

Pcg does not support profiling with pxp(1).

The following options have the same meaning as in pc (1), cc(1), and f77(1). See ld(1) for link-edit time options.

-c Suppress link-editing and produce '.o' file(s) from source file(s).

-g Generate additional symbol table information for sdb (which is obsolete).

-w Suppress warning messages.

-p Prepare object files for profiling; see prof(1).

-O Invoke an object-code optimizer.

-S Generate assembler code only; do not generate '.o' files.

-o name
Name the final output file 'name' instead of 'a.out'.

The following options are the same as in pc(1).

-C Compile code to perform runtime checks, and initialize all variables to 0.

-b Block buffer the file output.

The following options are peculiar to pcg.

-F Force the generation of new intermediate code, ignoring code maintained from previous compilations.

-d Generate debugging output.

FILES

file.p	Pascal source files
~saga/bin/epospcg	editor with resident symbol table component
~saga/bin/pcg	incremental recompilation driver
~saga/bin/pcgcodegen	portable C compiler intermediate code generator
/lib/f1	assembler generator
/usr/lib/pc2	inline expander
/usr/lib/pc3	separate compilation consistency checker
/lib/c2	peephole optimizer
/usr/lib/libpc.a	intrinsic functions and I/O library
/usr/lib/libm.a	math library
/lib/libc.a	standard library, see intro(3)
~saga/src/pcg/semantic	semantic phase sources
~saga/src/pcg/codegen	code generator sources
~saga/src/pcg/increm	incremental recompilation driver sources

SEE ALSO

"Pcg: A Prototype Incremental Compilation Facility for the SAGA Environment".
Berkeley Pascal User's Manual.

AUTHOR

John Kimball

BUGS

Pcg is a prototype system, and bug reports should be sent to the author.

BIBLIOGRAPHY

[Aho and Ullman] Aho, Alfred V. and Jeffrey D. Ullman. Principles of Compiler Design. Addison-Wesley, Reading, MA (1977).

[ANSI] Joint ANSI/X3J9-IEEE Pascal Standards Committee. An American National Standard: IEEE Standard Pascal Computer Programming Language. New York: Institute of Electrical and Electronics Engineers, Inc. (1983).

[Badger] Badger, Wayne H. "Make: A Separate Compilation Facility for the SAGA Environment," Master's Thesis, University of Illinois at Urbana-Champaign (1984).

[Beshers and Campbell] Beshers, George M, and Roy H. Campbell. "Maintained and Constructor Attributes." ACM SIGPLAN Symposium on Language Issues in Programming Environments (June 1985).

[Campbell and Kirslis] Campbell, Roy H. and Peter A. Kirslis. "The SAGA Project, a System for Software Development," Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, PA (April 1984).

[Cooper] Cooper, Doug. Standard Pascal User Reference Manual. W. W. Norton and Co., New York, NY (1983).

[Donzeau-Gouge, Huet, Kahn, and Lang] Donzeau-Gogue, Veronique, Gerard Huet, Gilles Kahn, and Bernard Lang. "Programming Environments Based on Structured Editors: The MENTOR Experience," INRIA Research Report No. 26, Rocquencourt, France (May 1980).

[Donzeau-Gouge, Lang, and Melese] Donzeau-Gogue, V., B. Lang, and B. Melese. "Practical Applications of a Syntax-Directed Program Manipulation Environment," Proceedings of the Seventh International Conference on Software Engineering, IEEE; Orlando, FA (March 1984).

[Estublier, Ghoul, and Krakowiak] Estublier, J., S. Ghoul, and S. Krakowiak. "Preliminary Experience with a Configuration Control System for Modular Programs," Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburg, PA (April 1984).

[Feldman] Feldman, S. I. "Make--A Program for Maintaining Computer Programs," Unix Programmer's Manual, Seventh Edition, volume 2; Bell Laboratories, Murray Hill, NJ (1980).

[Ganapathi and Fischer] Ganapathi, Mahadevan, and Charles N. Fischer. "Description-driven Code Generation using Attribute Grammars," Conference Record of the Ninth ACM Symposium on the Principles of Programming Languages, Albuquerque, NM (January 1982).

[Ghezzi and Mandrioli] Ghezzi, C. and D. Mandrioli, "Augmenting Parsers to Support Incrementality," Journal of the ACM, Vol. 27, No. 3 (July 1980).

[Goldberg] Goldberg, A. Smalltalk-80: The Interactive Programming Environment, Addison-Wesley, Reading, MA (1984).

[Habermann] Habermann, A. N. "An Overview of the Gandalf Project," CMU Computer Science Research Reviews (1980).

[Johnson1] Johnson, S. C. "A Tour through the Portable C Compiler," Unix Programmer's Manual, Seventh Edition, volume 2; Bell Laboratories, Murray Hill, NJ (1980).

[Johnson2] Johnson, S. C. "YACC: Yet Another Compiler-Compiler," Unix Programmer's Manual, Seventh Edition, volume 2; Bell Laboratories, Murray Hill, NJ (1980).

[Johnson and Fischer] Johnson, G. F., and C. N. Fischer. "Non-syntactic Attribute Flow in Language-based Editors," Conference Record of the Ninth ACM Symposium on the Principles of Programming Languages, Albuquerque, NM (January 1982).

[Joy, Graham, and Haley] Joy, William N., Susan L. Graham, and Charles B. Haley. "Berkeley Pascal User's Manual Version 3.0," Computer Science Division, Department of Electrical Engineering and Computer Science, University of California at Berkeley (July 1983).

[Joy and McKusick] Joy, William N., and M. Kirk McKusick. "Berkeley Pascal PX Implementation Notes Version 2.0," Technical Report, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California at Berkeley (January 1979).

[Kessler] Kessler, Peter B. "The Intermediate Representation of the Portable C Compiler, as used by the Berkeley Pascal Compiler," Technical Report, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California at Berkeley (April 1983).

[Kirslis, Terwilliger, and Campbell] Kirslis, Peter A., Robert B. Terwilliger, and Roy H. Campbell. "The SAGA Approach to Large Program Development in an Integrated Modular Environment," Proceedings of the GTE Software Engineering Environments for Programming-in-the-Large Workshop, Harwichport, MA (June 1985).

[Medina-Mora and Feiler] Medina-Mora, Raul, and Peter H. Feiler. "An Incremental Programming Environment," IEEE Transactions on Software Engineering, volume SE-7, number 5 (September 1981).

[Noonan and Collins] Noonin, R. E., and W. R. Collins. "The Mystro Parser Generator, PARGEN User's Manual, Version 6.3," College of William and Mary, Williamsburg, VA (1983).

[Paulson] Paulson, Lawrence. "A Semantics-directed Compiler Generator," Conference Record of the Ninth ACM Symposium on the Principles of Programming Languages, Albuquerque, NM (January 1982).

[Reiss] Reiss, Steven P. "PECAN: Program Development Systems that Support Multiple Views," IEEE Transactions on Software Engineering, volume SE-11, number 3 (March 1985).

[Reps] Reps, Thomas W. Generating Language-Based Environments. MIT Press, Cambridge, MA (1984).

[Richards] Richards, Paul G. "A Prototype Symbol Table Manager for the SAGA Environment," Master's Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign (1984).

[Teitelbaum and Reps] Teitelbaum, Tim, and Thomas Reps. "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment", Communications of the ACM, volume 24, number 9 (September 1981).

[Teitelman] Teitelman, Warren. "A Tour Through Cedar," Proceedings of the Seventh International Conference on Software Engineering, IEEE; Orlando, FA (March 1984).

[Teitelman and Masinter] Teitelman, Warren, and Larry Masinter. "The Interlisp Programming Environment," IEEE Computer, volume 14, number 4 (April 1981).

WRITING FILTER PROCESSES FOR THE SAGA EDITOR

Peter A.Kirslis

Department of Computer Science

Unviersity of Illinois at Urbana-Champaign

Urbana, Illinois

July, 1985

Writing Filter Processes for the SAGA Editor

Peter A. Kirslis

July, 1985

1. Introduction

The SAGA editor provides a mechanism by which separate processes can be invoked during an editing session to traverse portions of the parse tree being edited. These processes, termed *filter processes*, read, analyze and possibly transform the parse tree, returning the result to the editor. By defining new commands with the editor's user–defined command facility, which invoke filter processes, authors of filters can provide complex operations as simple commands. A tree plotter, pretty printer, and Pascal tree transformation program have already been written using this facility. This document introduces filter processes, describes parse tree structure and the library interface available to the programmer, and discusses how to compile and run filter processes. Examples are also presented to illustrate aspects of each of these areas.

2. The SAGA Editor

The SAGA editor is a language–oriented editor based upon a table–driven LALR(1) parser. As the user inputs his program, the editor analyzes the input and interactively builds a parse tree internally. Modifications are incrementally reparsed. Since the data is stored in parsed form, it is a simple matter to make the parse tree available for additional analysis by other programs. These programs, using pre–

defined library routines, can walk the parse tree collecting data. They can modify some fields in the tree directly, and can transform the structure of the tree by writing a text file which is passed back to the editor to be parsed and inserted in place of some portion of the existing tree. The editor provides both user–defined command sequences and command files to facilitate the use of these programs. See the SAGA editor user manual for more information about the editor itself.

3. The Parse Tree

The parse tree which is built by the editor consists of three types of parse tree nodes, and a header record. The node types consist of *terminal, non–terminal* and *marker*. The header record contains the root node of the tree, how many syntax or semantic errors are present, and other information. Each of these tree components is described in more detail in the following sections.

3.1. Parse Tree Structure

The root node of the tree is a non–terminal, and corresponds to the start symbol in the grammar defining the language in use. Each non–terminal node in the tree represents a non–terminal token on the left hand side of a production rule in the grammar. The children of each non–terminal node correspond exactly to the terminal and non–terminal tokens on the right hand side of this production rule. Each parent node points to its leftmost child; each child points to its right sibling (the rightmost child has no sibling); and each child points to its parent.

Each node also contains a *rightmost descendant* (or *rdescend* pointer). For terminal nodes, this descendant is the node itself. For non–terminal nodes, this node is the rightmost terminal node in this non–terminal's tree.

Terminal nodes are also linked together in a doubly–linked list, with each terminal node pointing to both the terminal node just preceding it and following it.

Each node also contains a *left thread* (or *lthread*) field, which points to the node which was on the top of the parse stack just before this node was shifted onto the stack. This field is used by the editor to reconstruct intermediate stages in the parse when a modification is being made to this portion of the tree.

3.2. Fields Common to all Nodes

In addition to the above mentioned link fields, each node also contains the parse state of the parser after this node was shifted onto the parse stack, a set of Boolean flags, some formatting information for printing, and integers which order the node relative to others around it.

Although the parse tables are not directly available, the editor module which provides access to them can be retrieved and added to the filter process. Queries concerning the parse states of nodes can then be made, for example, in a parse tree consistency checking program.

The following flags are available in each node:

FPOINT	An editor pointer is set at this node,
FDELETE	this node has been deleted from the parse tree,
FMODIFIED	this node is new to the parse tree since the last parse tree difference was taken,
FMAKE	reserved for use by the SAGA make facility,
FNOTPARSED	this node has not been parsed,
FSHIFTREDUCE	this node contains no parse state, since a shift–reduce action was performed by the parser,
FSELECT	used to highlight portions of the parse tree,
FSEMDELETE	this node has been deleted from semantic tables,
FLEXERR	this node contains a lexical error,
FSYNERR	this node contains a syntax error,
FSEMERR	this node contains a semantic error,
FELIDE	this node is being elided (not printed),
FSUBELIDE	this node is nested in an elision.

The FPOINT and FSELECT flags are only set during an editing session. FDELETE, FNOTPARSED, FSHIFTREDUCE, FLEXERR, and FSYNERR are manipulated by the parser. FMODIFIED is used by the tree–differencing facility. FMAKE is used by the SAGA make facility [Badger, 84]. FSEMDELETE and FSEMERR are manipulated by the semantic analysis routines. FELIDE and FSUBELIDE are intended to guide the printing (and hiding of detail) of te parse tree; they are not fully implemented yet.

Terminal nodes and non–terminal comment tree nodes contain *skipline* and *skipcol* fields to guide the printing of the node. The skipline field stores the number of newline characters to be output before the ascii string representing the token is printed, while the skipcol field stores the number of space characters to be output. The actual character string to be printed for the node can be retrieved with a call to one of the library routines to be presented later.

3.3. Terminal Nodes

Terminal nodes contain the token code of the terminal, a pointer to the print name of the terminal, and the length of the print name (not including preceding newlines and spaces). All relevant information described earlier is also present.

3.4. Non–terminal Nodes

Non–terminal nodes contain the token code of the non–terminal, the number of the production rule in the grammar for which this node represents the non–terminal on the left hand side of the production, and the leftmost child (first token on the right hand side of the production) of this node. All relevant information described earlier is also present.

3.5. Marker Nodes

If at any time during a parse, the parser encounters an error or an incomplete surrounding tree in its environment, the parser will suspend the parse. When it does so, it leaves a discontinuity in the tree. In order to be able to resume the parse at a later time, a marker node is inserted into the tree at the point of the error. This node stores a pointer to the node currently on the top of the parse stack, a pointer to the node which caused the error (if any), and a pointer to the "next" marker token in the parse tree. By next is meant the marker for the most recently occurring previous error. The first old terminal node following the new input (and any of its ancestors whose left thread pointer are identical) also have their left thread pointers reset to point to this marker node, so that later reparses that happen to reach this area of the tree will detect this discontinuity.

In general, it is not recommended that filter processes traverse parse trees containing discontinuities, since the tree structure will be incomplete. The fact that a parse tree has syntax errors (and semantic errors) can be detected by querying the *status*, *synerror* (and *semerror*) fields in the parse tree header record.

4. Filter Process Structure

Now that the reader has some idea about the structure of the parse tree, we will describe the structure of a filter process (program), and how it accesses the filter library of node access routines. The library itself will be described in the next section.

Two steps are necessary to use the filter library in a Pascal program. First, the filter library header file must be included. The Pascal program does this via the following *include* statement:

```
program myfilter(output);
#include "../../../src/filterlib/hdr/filterlib.h"
end. (* myfilter *)
```

The path given above assumes that the filter program resides in the filter process directory within the SAGA source code directory hierarchy. You will want to adjust it if the program source resides elsewhere.

This include file in turn references several include files used by the SAGA editor. These files define the interfaces used by the routines which access the parse tree nodes, many of which are used by the SAGA editor itself.

The include file which declares the routines which access the parse tree is *src/editor/hdr/nodeaccess.h.* The routines in this file are the ones described in the next section of this document.

The second step needed to use the filter library occurs at compilation time. The compiled object (relocatable binary) is linked and loaded together with the filter library as follows:

```
pc -c myfilter.p
pc -o myfilter myfilter.o ../../../src/filterlib/filterlib.
```

(The filter library may instead be stored in *saga/lib/filterlib.*) The compiled program may be used either in conjunction with an editing session, or stand alone on a parse tree file produced earlier by an editor. This latter ability can be helpful when debugging filter programs. When used with the editor, the filter process is invoked with some standard command line arguments, including the name of the directory containing the parse tree and related files. These conventions will be described in detail in a later section.

5. The Filter Library

The filter library consists of a number of functions and procedures which read (and some which modify) certain fields within nodes of the parse tree. These routines are divided into several categories, and presented in the following order: opening and closing the parse tree file, retrieving information from the header record, accessing pointers which connect nodes, accessing fields common to all node types, accessing fields specific to each node type, and modifying selected parse node fields.

5.1. Opening and Closing the Parse Tree Files

The following functions are provided to establish a connection to an existing parse tree file:

```
function Ninitialize          (* open a parse-tree directory *)
   (var pathname: charbuf;    (* name of directory *)
    var parsefile: filerange; (* return: file tag of parse tree *)
    var stringfile: filerange (* return: file tag of string table *)
   ): integer;                (* return 0 if o.k., -1 for error *)
   external;
```

The pathname parameter is the name of the directory containing the parse tree files. It is supplied to the filter process as the first argument on the command line; using the *argc* and *argv* Pascal system routines, this string can be retrieved and passed to *Ninitialize*. The second and third parameters are returned by *Ninitialize*, and are passed to other filter library routines.

Function *Nopen* is provided for completeness, should the filter process wish to define its own paged data structure, or open and reference a second parse tree file in addition to the first one opened above. If a second tree is to be accessed, both the parse tree file and string table file need to be explicitly opened. If only one tree is to be accessed, and no other files referenced, then this call need not be used.

```
function Nopen                (* open an existing paged file *)
   (var pathname: charbuf;    (* name of paged file *)
    var recsize: integer;     (* return: record size in bytes *)
    var recperpage: integer   (* return: records/page (for Nusebuffer()) *)
   ): integer;                (* return filetag if o.k., -1 if error *)
   external;
```

Note that the pathname parameter to *Nopen* refers to the actual file to be opened, not just the directory which contains the file. The remaining parameters are returned, and are to be passed to *Nusebuffer* to

assign to the file a buffer in which to place the file's data.

Nusebuffer assigns a buffer to the file to be paged into memory. The *bufaddr* parameter should be declared as a pointer to an array from 1 to n of records, where the record type is the record being paged. This pointer needs to be passed to the routine as ord(<pointer>) (so that *Nusebuffer* can be used for many record types). The *reccount* parameter specifies the number of records in the array, which must be an exact multiple of the page size returned by the *Nopen* call. If only *Ninitialize* called, *Nusebuffer* need not be called either, since the code in *Ninitialize* declares a buffer to contain the paged data, and also makes a call to Nusebuffer *itself.*

```
function Nusebuffer          (* assign data buffer to paged file *)
   (filetag: integer;        (* assign buffer to this file *)
    bufaddr: integer;        (* memory address of buffer (ord(b)) *)
    reccount: integer        (* record size of buffer *)
   ): integer;               (* return 0 if o.k., -1 for error *)
   external;
```

The *Nclose* routine should be called when the filter process is finished. If the parse tree file was only read, this call is not strictly necessary. However, if any fields were modified by the filter process, this routine must be called in order to write out the remaining data in memory and close the file, otherwise information may be lost.

```
function Nclose              (* close an open paged file *)
   (ft: filerange            (* file tag *)
   ): integer;               (* return 0 if o.k., -1 if error *)
   external;
```

5.2. Retrieving Information from the Header Record

These routines are provided to retrieve information from the header record: The most useful of these are *Nroot*, which returns the root node of the parse tree, and *Nstatus*, which returns the parse tree status, either COMPLETE or SUSPEND. COMPLETE will be returned only if the tree contains neither syntactic nor semantic errors. The

```
function Ndelete             (* get no. of explicitly deleted nodes *)
   (ft: filerange): nodeindex; external;
```

```
function Nmodified              (* get parse tree modified flag *)
  (ft: filerange): boolean; external;

function Nreadonly              (* get parse tree readonly flag *)
  (ft: filerange): boolean; external;

function Nroot                  (* get parse tree root node *)
  (ft: filerange): nodeindex; external;

function Nsemerror              (* get parse tree semantic error count *)
  (ft: filerange): integer; external;

function Nstatus                (* get parse tree status *)
  (ft: filerange): statuskind; external;

function Nsynerror              (* get parse tree syntax error count *)
  (ft: filerange): integer; external;

function Ntreesynlist           (* get parse tree .treesynlist pointer *)
  (ft: filerange): nodeindex; external;
```

5.3. Accessing Pointers which Connect Nodes

The parse tree nodes can be thought of as being stored as an array of nodes from 1 to n. Each node has an integer assigned to it which is used to reference it. This index is stored in and used by other nodes as well. These routines are presented below, with associated comments.

```
function Nf                     (* get next node on frontier of tree *)
  (ft: filerange; n: nodeindex): nodeindex; external;

function Nleftson               (* get leftmost child of non-term node *)
  (ft: filerange; n: nodeindex): nodeindex; external;

function Nlthread               (* get node beneath this one on "stack" *)
  (ft: filerange; n: nodeindex): nodeindex; external;
```

```
function Np                    (* get previous node on frontier of tree *)
   (ft: filerange; n: nodeindex): nodeindex; external;

function Nparent               (* get parent node *)
   (ft: filerange; n: nodeindex): nodeindex; external;

function Nrdescend             (* get rightmost terminal in this tree *)
   (ft: filerange; n: nodeindex): nodeindex; external;

function Nsibling              (* get right sibling *)
   (ft: filerange; n: nodeindex): nodeindex; external;
```

5.4. Accessing Fields Common to All Nodes

The following routines retrieve other information stored in each parse tree node. These fields are described in more detail in the parse tree description section earlier in this paper.

```
function Ndepth                (* get depth of node into tree *)
   (ft: filerange; n: nodeindex): integer; external;

function Nenum                 (* get ordering stamp of node *)
   (ft: filerange; n: nodeindex): integer; external;

function Nflagtest             (* test flag setting *)
   (ft: filerange; n: nodeindex; thisflag: short): boolean; external;

function Nnodetype             (* get type of parse tree node *)
   (ft: filerange; n: nodeindex): treenodetype; external;

function Npstate               (* get state of parser stored in this node *)
   (ft: filerange; n: nodeindex): staterange; external;

function Nskipcol              (* get skip column count for printing *)
   (ft: filerange; n: nodeindex): integer; external;
```

```
function Nskipline              (* get skip line count for printing *)
   (ft: filerange; n: nodeindex): integer; external;
```

5.5. Accessing Fields Specific to a Node Type

5.5.1. Fields Present in Terminal Nodes Only

Procedure *Nname* retrieves the print name of a terminal node. Both the parse tree file and string table file tags must be supplied to the routine. Calls to *Nskipline* and *Nskipcol* should also be made to retrieve the number of newlines and spaces to print before the token name, if these are needed.

```
procedure Nname                 (* get print name of token from string table *)
   (ftp, fts: filerange;        (* parse tree and string table file tags *)
    n: nodeindex;               (* node of interest *)
    var buf: charbuf;           (* return: print name of token *)
    var length: cbufindex       (* return: length of print name *)
   ); external;


function Ntoken                 (* get the token code of the terminal node *)
   (ft: filerange; n: nodeindex): tokenrange; external;


function Nvallength             (* get the length of the print name *)
   (ft: filerange; n: nodeindex): integer; external;
```

5.5.2. Fields Present in Non–terminal Nodes Only

The following routines are only meaningful when applied to non–terminal nodes. Note that a third routine *Nleftson*, mentioned earlier, is also only applicable to non–terminal nodes.

```
function Nntoken                (* get token code of non–terminal node *)
   (ft: filerange; n: nodeindex): tokenrange; external;


function Nrule                  (* get rule # of non–term node *)
   (ft: filerange; n: nodeindex): rulerange; external;
```

5.5.3. Fields Present in Marker Nodes Only

If the parser encounters an incorrect token during the parse, the parse will be suspended, a marker token inserted in the tree at this point, and the *badtoken* field of the marker set to point to this incorrect token. If, however, the parse is simply suspended (via a partial parse command in the editor for example), a marker will be inserted into the frontier of the parse tree at the point of the suspension, but no node will be assigned to the *badtoken* field.

```
function Nbadtoken          (* get offending node of marker node *)
  (ft: filerange; n: nodeindex): nodeindex; external;
```

Marker tokens are linked together in a list. The header record of the parse tree contains a pointer to the first marker token, and then each marker token contains a pointer to the next one. The *Nmarksynlist* routine is used to retrieve this pointer from a marker node.

```
function Nmarksynlist       (* get next error pointer in marker node *)
  (ft: filerange; n: nodeindex): nodeindex; external;
```

When a parse is suspended, the node on the top of the parse stack must be noted for later resumption of the parse. This node is stored in the *oldstacktop* field of the marker node, and can be retrieved by the *Noldstacktop* routine.

```
function Noldstacktop       (* get stack top stored in marker node *)
  (ft: filerange; n: nodeindex): nodeindex; external;
```

5.6. Modifying Selected Parse Node Fields

Presently, only the parse tree flags and format fields for printing of the nodes can be rewritten. Only those flags not maintained by the parser should be changed, or havoc will result. See the discussion of parse tree flags presented earlier in this paper for specific flag names.

The *skipline* and *skipcol* fields of the parse tree are used by the display manager to format the tree for printing. These may be reset to any appropriate non-negative value. A filter process to pretty print the parse tree would use these fields to reformat the tree. Note that both non-terminal and terminal nodes have these format fields, but only the nodes along the frontier of the tree have their formats read by the

display manager. Thus the format fields in internal nodes can be used to store formats as inherited attributes of the parse tree nodes. Coding a program in this manner could simplify the bookkeeping which would otherwise be necessary.

```
procedure Newflagclear        (* clear flag *)
   (ft: filerange; n: nodeindex; thisflag: short); external;


procedure Newflagset          (* set flag *)
   (ft: filerange; n: nodeindex; thisflag: short); external;


procedure Newskipline         (* set newline count to print before name *)
   (ft: filerange; n: nodeindex; value: integer); external;


procedure Newskipcol          (* set space count to print before name *)
   (ft: filerange; n: nodeindex; value: integer); external;
```

5.7. Accessing Other Specialized Data: An Array of Shorts

This next section presents one other type of record array which is predefined along with the parse tree node: an array from 1 to n of short integers (two bytes of storage per number). The following routine retrieves these shorts.

```
function Nptshort             (* get ptshort field *)
   (ft: filerange; st: integer): short; external;
```

6. Executing a Filter Process: Command Line Arguments

The SAGA editor contains a *filter* command which takes the name of the filter process as an argument, and arranges to execute the program as a sub–process to the editor. This command automatically supplies the name of the parse tree directory as the first argument to the program, and optionally supplies a parse tree node number as a second argument if a sub–tree is selected by the user to be passed to the filter command. Any other arguments given to the filter command are passed along to the filter process after these initial arguments. Thus the filter process is executed with the following arguments:

<filtername> <parse-tree-directory> [<tree-node>] [<args to filter cmd>]

When the filter process begins execution, it should first pass its first argument to *Ninitialize* to open the parse tree and string table files. If the optional second argument is present, it should be used as the starting node in the tree to be processed. If it is absent, a call to *Nroot* will return the root node of the parse tree, which should be used instead.

Unless the process is prepared to deal with discontinuities in the parse tree, it is a good idea to call *Nstatus*, *Nsynerror* and/or *Nsemerror* to determine whether any exist. If this is the case, the process may wish to simply produce an error message and exit.

If a sub-tree has been specified to the filter process and discontinuities exist in the parse tree, it is possible to determine whether any exist within the subtree of interest. One approach is to traverse the frontier of the subtree, checking for the presence of a marker node or a node with the FNOTPARSED flag set. Alternatively, the *Ntreesynlist* and *Nmarksynlist* routines can be called to retrieve the first and successive marker tokens in the tree, respectively. The *Nenum* routine could check the enumeration field of each of these marker nodes or the *Nbadtoken* node associated with the marker to see whether it is in between the enumeration fields of the first and last terminal nodes in the sub-tree of interest. If none are found, the processing can go ahead.

7. Traversing the Parse Tree

Once the files are opened and the tree status determined, the *Nleftson* and *Nsibling* routines can be used to perform a pre-order, in-order, or post-order walk of the parse tree. Alternatively, starting at the first terminal node in the tree, the *Nsibling* and *Nparent* routines can be used to walk the tree in the same order as the canonical parse which constructed it. Starting at the first terminal node, the *Nf* routine could also be used to walk the frontier of the tree.

At each node in the tree, the appropriate library routine can be used to retrieve the fields of interest in the node.

Should it be desired to make modifications to the tree, two approaches may be used. Fields such as the skipline and skipcol fields can be queried and reset directly using the *Newskipline* and *Newskipcol* routines. To transform the tree, a text file should be created into which the new text to be inserted into the tree is placed. If the *filter* command in the editor is placed into a user–defined command sequence, then additional commands in this sequence can cause the deletion of the sub–tree which was passed to the filter followed by the insertion of the new text from this file.

For more complex modifications, the filter process can created a command file which contains a combination of editor commands and input data. The user–defined command sequence which executes the filter command can then invoke the editor's *exec* command on the file produced by the filter process; commands in this file will then guide the modifications to be made.

Note that if the filter process plans to modify the parse tree in any way, the filter command in the editor should be given as *filter -w ....* The –w option tells the editor to close the parse files and re–open and re–read their contents once the filter process has completed. Normally, all data in memory is written to disk before the filter process is invoked, but a copy is kept in memory for efficiency, and is reused when the editor continues execution.

8. Summary

This document has described the implementation of filter processes. Constructive comments, questions, and feedback concerning unclear or incomplete sections should be be directed to the author.

MANUAL PAGES FOR SAGA SOFTWARE TOOLS

Carol S. Beckman

George Beshers

David Hammerslag

Peter A.Kirslis

Hal Render

Robert Terwilliger

Department of Computer Science

University of Illinois at Urbana-Champaign

Urbana, IL

July, 1985

NAME

dfbase—set the base version for finding differences between SAGA parse trees

SYNOPSIS

dfbase <saga directory>

DESCRIPTION

Dfbase sets the base version for dfdiff to use. The saga directory contains the files created by epos. The modified fields in the current files are cleared and the parse tree is copied to the base version. The parse tree may not become the base version if it contains errors or parse suspension points.

DIAGNOSTICS

Error messages are (hopefully) self-explanatory.

FILES

In the saga directory for which dfbase is invoked: dfbaseparse parse tree for base version dfbasestr string file for base version dfdebug debugging output saga1p parse tree for the version being edited saga1s string file for the version being edited

SEE ALSO

dfdiff, dfundo

IDENTIFICATION

Carol Beckman

BUGS

Dfbase will change in the near future with little notice.

NAME

dfdiff—display differences between SAGA parse trees

SYNOPSIS

dfdiff <saga directory> [<root or range>] [<context>] [<version>]

DESCRIPTION

Find the differences between the current version of the parse tree and an older, base, version.

The <root or range> argument tells which differences to print. If the argument is an integer, it is taken as the root (nodeindex) of a subtree. If the argument is two integers separated by a colon, it is taken as the beginning and ending locations (nodeindices) of the range in which to find differences. Only differences in the selected part of the parse tree are printed. If no argument is given, all the differences in the tree are printed.

The <context> argument tells how many lines of context to print around each difference. <context> is an integer. A partial line adjacent to a difference counts as one line. If no argument is given, one is used.

The <version> argument is used to select the version of the difference command. <version> is an integer. Currently only one version is available. This version is used if no <version> argument is given.

Dfdiff operates in screen mode or line mode. In line mode the differences will all print with no further input from the user.

In screen mode, the differences are displayed one at a time. If a difference cannot fit on one screen, the old and new parts of the difference each get half the space. The text can be scrolled so that all the difference can be viewed. Control–L scrolls the parts forward, while control–H scrolls back. The old and new parts can be scrolled individually by prefixing the command with control–O or control–N for the old and new parts, respectively. So control–O control–L scrolls just the old part forward. Control–N control–H scrolls just the new part back. Moving from one difference to the next is accomplished with control–J and control–K. Control–J moves to the next difference. Control–K moves back one difference. The default action is to move to the next difference. So if any other key is hit, the next difference is displayed.

DIAGNOSTICS

Error messages are (hopefully) self–explanatory.

FILES

In the saga directory for which dfdiff is invoked: dfbaseparse parse tree for base version dfbasestr string file for base version dfdiffinfo information for the differences found dfdebug debugging output saga1p parse tree for the version being edited saga1s string file for the version being edited

SEE ALSO

dfbase, dfundo

IDENTIFICATION

Carol Beckman

BUGS

When called as a filter command from the SAGA editor, the first screen display is not always correct. This affects further screen displays since only the new text is plotted and the replotter assumes the first screen was properly displayed. This might be fixed now.

The field in the parse tree which is supposed to indicate whether a change has been made since the last time dfdiff was executed does not get set by all changes. Thus dfdiff may not display the new changes since it will reuse the old information, on the false assumption that it is current.

If dfundo is used to undo differences, but these differences are not actually undone, dfdiff will not display the undone differences unless the parse tree is modified.

NAME

dfundo—generate commands for undoing differences between SAGA parse trees

SYNOPSIS

dfundo <saga directory> <diff#> ... <diff#>

DESCRIPTION

Dfundo generates the commands needed to undo a difference. Dfdiff must have been executed after any changes to the parse tree and before dfundo is invoked. The <diff#>s are the numbers given by dfdiff of the differences which are to be undone. One or more <diff#>s may be given for one invocation of dfundo.

DIAGNOSTICS

Error messages are (hopefully) self-explanatory.

FILES

In the saga directory for which dfundo is invoked: dfbaseparse parse tree for base version dfbasestr string file for base version dfdiffinfo information for the differences found dfdebug debugging output saga1p parse tree for the version being edited saga1s string file for the version being edited #dfundoexec file of commands to execute to undo differences

SEE ALSO

dfdiff, dfbase

IDENTIFICATION

Carol Beckman

BUGS

The commands generated by dfundo cannot be executed by epos with an exec command. It seems that epos interprets the text for insertions as commands. The range syntax needed for the deletions is not implemented.

Dfundo will report that a difference has been undone already even if the file of commands has not be executed unless some change is made to the parse tree and dfdiff is executed again.

NAME

epos — language–oriented editor based on an LR(1) parser.

SYNOPSIS

epos [–l] [–P<parse–tables>] [–cdiimprstvx] <parse–tree> [<parse–tree>]

DESCRIPTION

Epos is an editor for languages based on formal BNF style grammars and LR(1) parsers. An editor can be produced for any language for which such a description exists. The editor provides both text–oriented commands and additional structure–oriented commands, which are based on the structure of the parse tree produced by the editor.

The editor incorporates an LR(1) style parser to perform syntactic and optional semantic analysis of the program being edited. Each time the user completes an insertion or modification, the parse tree is incrementally updated with the new information. The user of the editor is provided with additional analysis during the editing process, and presented with immediate feedback about the correctness of the input.

The amount of semantic analysis performed (and whether any at all occurs) is dependent both on the parser–generating system used to produce the editor, and the type of semantic analysis defined in the input grammar file.

The editor is screen–oriented, using the *termcap* facility to adapt itself for a particular terminal; a line mode is also provided. The SAGA editor user manual provides a description of editor commands. Information about the run–time environment of the editor, and its command line options and arguments is presented here.

The command line options are:

–l Invoke the editor in line mode instead of screen mode.

–P Specifies an alternate file (–P<parse–tables>) from which to load the parse tables to be used.

Since the editor is still an experimental prototype, a number of the available debugging options are listed below to aid the individuals managing the implementation. These options can be activated either by command line flags or the *on* and *off* commands of the editor. Users might find them useful in formulating bug reports. The command line options for debugging are:

–b Turn on paging system debugging. Same as the "on db" editor command. If specified twice, also enables detailed debugging.

–c Turn on command interpreter debugging. Same as "on dc".

–i Turn on input and editor initialization debugging. Same as "on di". If specified twice, also enables detailed debugging.

–m Turn on make (incremantal recompilation) system debugging. Same as "on dm".

–p Turn on parser debugging. Same as "on dp".

–r Turn on parser initialization and recovery debugging. Same as "on dr".

–s Turn on debugging of the semantic analysis phase of the parse. Same as "on ds".

–t Turn on debugging of the parse tables (used in the editor's language dependent module only). Same as "on dt".

–x Turn on debugging of the lexical analysis phase of the parse. Same as "on dl".

FILES

saga/bin/epos:
cshell script to invoke the editor,
saga/obj/editor/<language>.mystro/epos:

the actual editor process,
saga/obj/editor/<language>.mystro/parse.tables:
the binary parse tables,
saga/obj/editor/<language>.mystro/help.index:
index to on-line help file,
saga/obj/editor/<language>.mystro/epos.help:
on-line help file,
saga/src/editor/lib/epos.cmds:
user-defined commands for all editors,
saga/src/editor/lib/epos.<language>cmds:
user-defined commands for this language,
<current-directory>/.epos.<language>cmds:
the user's private user-defined commands for this language.

SEE ALSO

scat(1), dfbase(1), dfdiff(1), dfundo(1), rulecount(1).

AUTHOR

Peter A. Kirslis, Dept. Computer Science, Univ. Illinois — Urbana, 1304 W. Springfield Ave., Urbana, Illinois, 61801. Written 1982, revised and extended 1983, 1984, 1985.

BUGS

The editor is still an experimental prototype. Some bugs still exist in the parser, although most problems will be found in the screen-mode command interpreter. If a parse tree file is garbled by the editor, its text representation can usually be recovered with the *scat(1)* command.

The second parse tree argument to the editor specifies an alternate parse tree to be accessed read-only. Use of the alternate file is restricted to line mode, since the screen mode interpreter does not yet provide any support for accessing it.

Multi-line comments are not yet supported in the editor. The lexical analyzer does recognize them and store them properly, but the command interpreters and screen display do not yet handle them properly.

NAME

Make – maintain program groups

SYNOPSIS

Make [**–f** makefile] [option] ... file ...

DESCRIPTION

Make executes commands in *makefile* to update one or more target *names*. *Name* is typically a program. If no **–f** option is present, 'makefile' and 'Makefile' are tried in order. If *makefile* is '–', the standard input is taken. More than one **–f** option may appear

Make updates a target if it depends on prerequisite files that have been modified since the target was last modified, or if the target does not exist.

Makefile contains a sequence of entries that specify dependencies. The first line of an entry is a blank–separated list of targets, then a colon, then a list of prerequisite files. Text following a semicolon, and all following lines that begin with a tab, are shell commands to be executed to update the target. If a name appears on the left of more than one **:** then it depends on all of the names on the right of the colon on those lines, but only one command sequence may be specified for it. If a name appears on the left of a colon exclamation mark **:!** then it depends on exactly one of the files on the right of the colon exclamation mark. The file choosen is the first one (left to right) that exists, or the last one if none of them exists. If a name appears on the left of a colon question mark **:?** then it depends on all the files on the right of the colon question mark *if* they exist. If a name appears on the left of a colon exclamation question mark **:!?** then it depends on no more than one of the files on the right, if no file on the right exists, then it behaves like a **:?** . If a name appears on a line with a double colon **::** then the command sequence following that line is performed only if the name is out of date with respect to the names to the right of the double colon, and is not affected by other double colon lines on which that name may appear.

Three special forms of a name are recognized. A name like $a(b)$ means the file named b stored in the archive named a. A name like $a((b))$ means the file stored in archive a containing the entry point b. Also a name like $a,v(b)$ refers to the RCS file of a with revision b. The revision may contain symbolic names as defined in RCS. If the revision refers to a branch then the last member of that branch is the revision chosen. Note: Using the modified *ci* command with –l or –u options the modification dates of a revision and the working file are equal, i.e., neither one is considered to be out of date with the other.

Sharp and newline surround comments.

The following makefile says that 'pgm' depends on two files 'a.o' and 'b.o', and that they in turn depend on '.c' files and a common file 'incl'.

```
pgm: a.o b.o
        cc a.o b.o –lm –o pgm
a.o: incl a.c
        cc –c a.c
b.o: incl b.c
        cc –c b.c
```

Makefile entries of the form

string1 = string2

are macro definitions. Subsequent appearances of $(*string1*) or ${*string1*} are replaced by *string2*. If *string1* is a single character, the parentheses or braces are optional.

The value of a macro may be edited before being replaced in the input stream. The syntax is ${string1:modifier} where *modifier* specifies the edit to be made. If an edit fails a default value is returned and a warning is sent to stderr. The modifiers are:

-a Which returns the archive file. Thus dir1/archive(member) becomes dir1/archive. If no (exists then the argument is returned.

-e Which returns the extension if one exists or .junk otherwise. Thus ../dir1/root.e1.e2 becomes .e2.

-h Which returns the head of the path name if a / exists in the argument, otherwise it returns a '.' (current directory). Special case, if the path is the root name / then that is returned. Thus dir1/dir2/name becomes dir1/dir2.

-m Which returns the member of an archive if a (exists, otherwise it returns its argument.

-R **-R/.E/** The first case returns the "local" root of the path name, i.e., all the directories and the extension are discarded. The second case appends the new extension to the former result. Thus dir1/dir2/name.e becomes name.

-r **-r/.E/** This version retains the directories. In the example dir1/dir2/name is returned.

-t Which returns the tail of the path name if a / exists or its argument otherwise.

-s Which implements the Unix ed command s/pattern/replace/. If the pattern match fails the argument is returned.

All of the modifiers work on lists of names by processing each name individually, i.e., the strings are broken into lists of names based on space delimiters and each name is modified separately.

For each rule four special variables are set, $@, $*, $<, and $?. The special macro $@ stands for the full target name, $* stands for the target name with the suffix deleted. Both of these variables may be used in the prerequisites list and the commands in conjunction with the editing operations explained above. The macro $< lists the prerequisites that exist on the line with the commands, and $? lists all the prerequisites that are out of date. The special variables can be used with the modifiers discussed above.

Shell meta characters can occur in both target and prerequisite file names. When used in target file names the pattern is used to find the rules associated with an actual target name. When a match occurs the $@ and $* variables are set to the actual target name, and the prerequisites are processed. If a prerequisite contains a meta character the corresponding directory is searched and any file which matches becomes an actual prerequisite. The standard glob(1) patterns have been extended with the ** pattern which is like * but capable of matching a sequence of directories when used in the target name.

Make can infer prerequisites for files for which the *Makefile* gives no explicit commands. For example, a '.c' file may be inferred as prerequisite for a '.o' file and be compiled to produce the '.o' file. Thus the preceding example can be done more briefly:

```
pgm: a.o b.o
        cc a.o b.o -lm -o pgm
a.o b.o: incl
```

Prerequisites are inferred from a list of optional rules. Optional rules are distinguished by a :? between the targets and dependent files. The optional rules only apply if the dependent file(s) exists, and only one optional rule applies for a particular target. Thus order is significant; the commands associated the first target pattern that matches target name and for which there exists a dependent file are the commands used. For example, the rule for making optimized '.o' files from '.c' files is

```
*.o :? $*.c
cc -c -O -o $@ $*.c
```

Notice the use of a shell meta character in the target file name, and the special macro $* to specify the exact prerequisite desired.

Certain macros are used by the default inference rules to communicate optional arguments to any resulting compilations. In particular, 'CFLAGS' is used for *cc*(1) options, 'FFLAGS' for *f77*(1) options, 'PFLAGS' for *pc*(1) options, and 'LFLAGS' and 'YFLAGS' for *lex* and *yacc*(1) options. In addition, the macro 'MFLAGS' is filled in with the initial command line options supplied to *make*. This simplifies maintaining a hierarchy of makefiles as one may then invoke *make* on makefiles in subdirectories and pass along useful options such as -k.

Command lines are executed one at a time, each by its own shell. A line is printed when it is executed unless the special target '.SILENT' is in *makefile,* or the first character of the command is '@'.

Commands returning nonzero status (see *intro*(1)) cause *make* to terminate unless the special target '.IGNORE' is in *makefile* or the command begins with <tab><hyphen>.

Interrupt and quit cause the target to be deleted unless the target is a directory or depends on the special name '.PRECIOUS'. All files ending in ,v or having the form ,v() are assumed to be precious.

Other options:

-i Equivalent to the special entry '.IGNORE:'.

-k When a command returns nonzero status, abandon work on the current entry, but continue on branches that do not depend on the current entry.

-n Trace and print, but do not execute the commands needed to update the targets.

-t Touch, i.e. update the modified date of targets, without executing any commands.

-r The predefined macros and default rules are not processed which saves processing time, and protects the user from hidden intertactions. The special entry '.NORULES:' is equivalent.

-s Equivalent to the special entry '.SILENT:'.

-q Test the prerequisites of a (single) target, and return a 0 status if the target is up to date and -1 status if it needs to be remade.

-Q For recursive calls to make asking for the special status reports of -q. Notice that a positive status indicates an error in the child make.

The most common use of make is in maintaining large programs. In the following example all the *.p* files are stored in the directory *../src* and all the *.h* are stored in the directory *../hdr* and the objects are going to be placed in this directory.

```
SrcDir  = ../src
Srcs    = program.p module1.p module2. module3.p
Objs    = ${Srcs:r,.o,}
program : ${Objs}
        ${PC} ${PFLAGS} ${Objs} -o program
${Objs} : ${SrcDir}/$*.p
        ${PC} ${PFLAGS} -c $<
${Objs} : ../hdr/*.h
```

Notice that the object names were generated with the modifier *r*. The second rule should be considered a *foreach* object file generate the specified prerequisite and Pascal compile. The third rule specifies that all the objects are dependent on all the headers.

We present two examples of using make to maintain RCS files. (Macros as defined above).

```
Rev     = working
```

```
RcsFiles = ${Srcs:s,.*,RCS/&,v(${Rev}),}
All      : ${RcsFiles}
${RcsFiles} : $*.p
        ci -u${Rev} $<
```

After you are done editing the working files this make script automatically discovers which files were actually touched, and checks them in. Note the use of a symbolic revision name.

```
program : ${Objs}
        ${PC} ${PFLAGS} ${Objs} -o program
${Objs} :? $*.p
        ${PC} ${PFLAGS} -c $<
${Objs} :? ${SrcDir}/$*.p
        ${PC} ${PFLAGS} -c $<
**.p : ${SrcDir}/RCS/${@:t},v(working)
        ${CO} -r${Rev} $@ $<
```

This example searches two directories for the Pascal sources, first the current directory, and then the SrcDir. However both sets of sources are dependent on the same RCS files.

An example of archive maintainance is

```
SRCDIR=       ../src
INCLUDE = /usr/include
SRCS=open.c  close.c  creat.c
archive.a:        ${SRCS:s,^).c$,system.o(1.o),}
                  ar rv archive.a ${?:m}
                  rm ${?:m}
                  ranlib archive.a
archive.a: ${INCLUDE}/system.h
archive.a(*.o):? ${@:m}
                  echo Using ${@:m}
*.o:      ${*:s,.*,${SRCDIR}/&.c,}
                  ${CC} ${CFLAGS} $<
archive.a(*.o):?          ${${@:m}:s,).o,${SRCDIR}/1.c,}
                  ${CC} ${CFLAGS} $<
Maketd:
                  Maketd -mMakefile -Asystem.o -s${SRCDIR} ${SRCS}
```

Notice that the **ar** command is executed once with all the **.o** files which are out of date, avoiding some overhead.

The macro ${MAKE} is recognized as the current make command, and treated specially. It is called with ${MFLAGS} as arguments, and also called when the –n option is in effect. When Make is called from Make a return code is requested and examined to see if the target was remade.

FILES

makefile, Makefile

SEE ALSO

sh(1), touch(1), f77(1), pc(1), Maketd(1)

BUGS

Some commands return nonzero status inappropriately. Use –**i** to overcome the difficulty.
Commands that are directly executed by the shell, notably *cd*(1), are ineffectual across newlines in *make*.

NAME

rulecount — a SAGA parse tree analyzer

SYNOPSIS

rulecount [options] countfile [sagafile ...]

DESCRIPTION

Rulecount is a program which counts the uses of production rules in a SAGA parse tree. A report is produced on the standard output giving the indices of the rules found and their corresponding multiplicity. Various options may be invoked to produce different reports. The counts are stored in the file given as the countfile on the command line, and these counts can be accumulated over several runs of the program. This allows one, for example, to run the program with a test suite for a given set of editor files and determine whether all rules have been used or, if not, which ones have not. Each *sagafile* is a directory produced by a SAGA language-oriented editor, and from 0 to 32 files may be given on the command line. If no *sagafile* is given, the *countfile* is analyzed and a summary report is produced for the values stored in it.

Rulecount first performs a traversal on the SAGA parse tree file from an input SAGA editor directory, saving the counts of the rules used in the *countfile*, either creating a new file if one does not exist, or adding the counts to the *countfile* if one does exist. The program performs a traversal on each SAGA parse tree file on the command line, accumulating the results in the countfile. On completion of all the traversals, a summary report is produced for the accumulated counts, including the counts which existed, if any did, in the *countfile* when the program was run. Various options can be used to control the analysis and the report produced:

-o*N* inform *rulecount* of the index, *N*, of the origin rule of the grammar which the particular SAGA editor used in producing the parse tree file.

-r*N* inform *rulecount* of the index, *N*, of the maximum rule of the grammar which the particular SAGA editor used in producing the parse tree file.

-r*N* include in the output report only those rules which occurred *N* or more times in the input file. This defaults to 1 if this option is not used.

-i generate a report for each SAGA file in addition to the summary report which is always produced. This allows one to see which files used which rules. A few additional statistics are included in the individual reports, such as a count of the nodes and their types as found in each SAGA file, as well as the maximum depth reached in the traversal stack. This last value may be used to gauge the depth of the parse tree.

-p print the percentage of the grammar rules used in a particular parse tree. To use this option, the -o and -r options must also be used (for obvious reasons). If the -i option is on, the percentage used by each parse tree as well as the total percentage covered by all are reported.

-z display only those rules which have not been used (have a count of zero). It is recommended that the -r and -o options be turned on when using this, so that the program knows what the upper and lower bounds of the grammar rules are. Otherwise, it only gives those rules which lie between the current minimum and maximum rules found.

-t trace the traversal of the SAGA parse trees. This is primarily a debugging option, and is recommended only as a last resort, as it produces scads of output (a single line for each node of a parse tree).

-h display the usage line and the list of available options for the program. This information is stored in the file 'help.rulecount' in the saga/src directory containing the program source.

DIAGNOSTICS

Errors in the arguments to rulecount are flagged, and conditions which violate the integrity of the report are also checked, such as the occurrence of a rule whose index is greater than that given in the -r option. Most of these errors cause the program to halt immediately. As intermediate counts are written out to the countfile after each parse tree has been traversed, the contents of the countfile may be corrupted by spurious input. Some attempts have been made to indicate where the error occurred, thoUGH these may not always be sufficient for full debugging.

FILE MODES

The user must have read/write permission on the countfile and read permission on the SAGA file(s) on the command line.

FILES

~saga/bin/rulecount -- the executable program file ~saga/src/utilities/rulecount — the source directory ~saga/lib/help.rulecount -- the help file

IDENTIFICATION

The author of this program was Hal Render, currently working for the University of Illinois. All problems and suggestions for improvement should be addressed to him. His current address is:

Hal Render
222 Digital Computer Lab
University of Illinois
1304 W. Springfield
Urbana, Illinois 61801
(217) 333-7937

BUGS

The program does not currently check to see if the input SAGA files come from the same editor or even the same language. The user must take care not to mix files from different editors or languages, if he/she wishes an accurate report on the parse tree files. This program has not been tested very rigorously, and is thus subject to error. If any problems are found, please contact Hal Render.

NAME

scat – catenate and print the text from SAGA parse tree directories.

SYNOPSIS

scat <parse-tree-directory> [<parse-tree-directory> ...]

DESCRIPTION

Scat produces the source text representation of a SAGA parse tree on standard output. If more than one parse tree is specified, the output will contain the text from each tree, in the order that the arguments were supplied. Scat operates by traversing only the frontier of the parse tree, so it may be used to extract the text from parse trees containing discontinuities (suspension points and errors). It also can recover the text from parse trees whose internal structure has been scrambled, as long as the frontier is intact (which is usually the case when a parser bug in the editor occurs).

SEE ALSO

epos(1)

AUTHOR

Peter A. Kirslis, Dept. Computer Science, Univ. Illinois -- Urbana, 1304 W. Springfield Ave., Urbana, Illinois, 61801. Written February, 1985.

NAME

sem_create - create a semaphore

SYNOPSIS

~saga/bin/sem_create semaphore_name

DESCRIPTION

sem_create creates a semaphore to control interprocess communication. The semaphore is implemented with a file. To create a semaphore, execute sem_create and provide a name for a semaphore. The name of the semaphore should have the suffix **.sem.** sem_create creates a file named semaphore_name.

DIAGNOSTICS

sem_create will print an error message if more than one argument is given or if the argument does not end with .sem.

SEE ALSO

sem_intro(1), sem_destroy(1), sem_p(1), and sem_v(1). A C interface is described in sem_C_int(2).

IDENTIFICATION

Bob Terwilliger, UIUC DCL Urbana, Ill. 61801. Phil Roberts, UIUC DCL Urbana, Ill. 61801.

NAME

sem_destroy - destroy a semaphore

SYNOPSIS

~saga/bin/sem_destroy semaphore_name

DESCRIPTION

sem_destroy destroys a semaphore. To destroy a semaphore, execute sem_destroy with the semaphore name as the only argument. The name of the semaphore should have the suffix **.sem.**

DIAGNOSTICS

sem_destroy will print an error message if more than one argument is given or if the argument does not end with .sem.

SEE ALSO

sem_intro(1), sem_create(1), sem_p(1), and sem_v(1). A C interface is described in sem_C_int(2).

IDENTIFICATION

Bob Terwilliger, UIUC DCL Urbana, Ill. 61801. Phil Roberts, UIUC DCL Urbana, Ill. 61801.

NAME

sem_p - perform a P operation on a semaphore

SYNOPSIS

~saga/bin/sem_p semaphore_name

DESCRIPTION

sem_p performs a P operation on a semaphore. If a P operation has already been performed on the semaphore, the new P operation will block. The name of the semaphore should have the suffix **.sem.** The P operation is performed in the following manner. An flock is performed on the file that represents the semaphore (the file is created by sem_create). If a P operation has already been performed, the flock will block. The process now attempting the P will remain blocked until the process holding the flock is killed.

When the flock succeeds, a new process is forked to hold the flock. The PID of the new process is written in the semaphore file and the process goes to sleep. The corresponding V operation reads the PID from the semaphore file and kills the process holding the flock allowing the next process to perform its P operation.

DIAGNOSTICS

sem_p will print an error message if more than one argument is given or if the argument does not end with .sem.

SEE ALSO

sem_intro(1), sem_create(1), sem_destroy(1), and sem_v(1). A C interface is described in sem_C_int(2).

IDENTIFICATION

Bob Terwilliger, UIUC DCL Urbana, Ill. 61801. Phil Roberts, UIUC DCL Urbana, Ill. 61801.

NAME

ted, browse, peg – a family of prototype tree structure editors

SYNOPSIS

ted [<filename>]
browse [<filename>]
peg [<filename>]

DESCRIPTION

These are a family of closely related editors for editing unrestricted trees. Each of these editors is unique, although they share a common editor core and common editing features. Each editor consists of the (slightly tailored) editor core, and packages of external programs that operate on the tree constucted by the editor. The basic paradigm of ted editing is: the user constructs or modifies trees using the editor, then from within the editor, invokes external programs to certify that the tree maintains its desired properties. The user is encouraged to create his own external programs to suit his particular needs.

DIAGNOSTICS

Ted–based editors are chocked full of self–explanatory error messages.

FILES

.tedrc ted initialization file (lisp commands)

SEE ALSO

Since the ted editors are prototypes, they are rapidly changing; however the most comprehensive document is "Ted: a Tree Editor with Applications for Theorem Proving", by David Hammerslag. The uiucdcs local notesfile "ted" is a good source for up–to–date (tho less comprehensive) information.

IDENTIFICATION

David Hammerslag uiucdcs!hammer

BUGS

Being prototypes these editors are problably loaded with bugs.

There is very little hard documentation on any of the editors except ted.

NAME

sem_create - create a semaphore to control access to a file

SYNOPSIS

#include "~saga/src/sem_C_int/sem_C_int.h" #include "~saga/src/msc/msc.h"

int rtrn ;

int sem_create(file_name,semaphore,argc,argv) char file_name[] ; char semaphore[] ; int argc ; char *argv ;

cc * ~saga/src/sem_C_int/sem_C_int.o ~saga/src/sem_C_int/msc.o

DESCRIPTION

sem_create creates a semaphore to control access to a file. The semaphore controls access to **file_name. semaphore** receives the name of the semaphore when sem_create is done. The name of the semaphore is **file_name** with **.sem** concatenated to the end. **sem_create** executes the system program ~saga/bin/sem_create to create the semaphore. **semaphore** is the name of the file used for the semaphore. In other words, this function executes the command "sem_create semaphore".

DIAGNOSTICS

rtrn gets the return code from the system call to execute sem_create.

SEE ALSO

sem_create(1), sem_destroy(2), sem_p(2), sem_v(2).

IDENTIFICATION

Bob Terwilliger, UIUC DCL Urbana, Ill. 61801. Phil Roberts, UIUC DCL Urbana, Ill. 61801.

NAME

C interface to semaphore routines.

SYNOPSIS

#include "~saga/src/sem_C_int/sem_C_int.h" #include "~saga/src/msc/msc.h"

int rtrn ;

int sem_destroy(semaphore,argc,argv) char semaphore[] ; int argc ; char *argv ;

cc * ~saga/src/sem_C_int/sem_C_int.o ~saga/src/sem_C_int/msc.o

DESCRIPTION

sem_destroy destroys the semaphore created by The argument **semaphore** is the name of the semaphore created when sem_create(2) was called.

DIAGNOSTICS

rtrn contains the return code from the system call.

SEE ALSO

sem_C_int(2), sem_create(2), sem_intro(1), sem_create(1), sem_destroy(1).

IDENTIFICATION

Bob Terwilliger, UIUC DCL Urbana, Ill. 61801. Phil Roberts, UIUC DCL Urbana, Ill. 61801.

NAME

sem_p - perform a P operation on a semaphore

SYNOPSIS

#include "~saga/src/sem_C_int/sem_C_int.h" #include "~saga/src/msc/msc.h"

int rtrn ;

int sem_p(semaphore,argc,argv) char semaphore[] ; int argc ; char *argv ;

cc * ~saga/src/sem_C_int/sem_C_int.o ~saga/src/sem_C_int/msc.o

DESCRIPTION

sem_p performs a P operation on semaphore. The function really executes the command "sem_p semaphore". A V operation can be performed on the semaphore by calling sem_v(2). **semaphore** is the name of the semaphore created by calling sem_create(2).

DIAGNOSTICS

rtrn contains the return code from the call to system.

SEE ALSO

sem_C_int(2), sem_v(2), sem_intro(1), sem_p(1), sem_v(1).

IDENTIFICATION

Bob Terwilliger, UIUC DCL Urbana, Ill. 61801. Phil Roberts, UIUC DCL Urbana, Ill. 61801.

NAME

sem_v - perform a V operation on a semaphore

SYNOPSIS

#include "~saga/src/sem_C_int/sem_C_int.h" #include "~saga/src/msc/msc.h"

int rtrn ;

int sem_v(semaphore,argc,argv) char semaphore[] ; int argc ; char *argv ;

cc * ~saga/src/sem_C_int/sem_C_int.o ~saga/src/sem_C_int/msc.o

DESCRIPTION

sem_v performs a V operation on semaphore. The function really executes the command "sem_v semaphore". A P operation can be performed on the semaphore by calling sem_p(2). **semaphore** is the name of the semaphore created when sem_create(2) was called.

DIAGNOSTICS

rtrn contains the return code from the call to system.

SEE ALSO

sem_C_int(2), sem_p(2), sem_intro(1), sem_p(1), sem_v(1).

IDENTIFICATION

Bob Terwilliger, UIUC DCL Urbana, Ill. 61801. Phil Roberts, UIUC DCL Urbana, Ill. 61801.

NAME

Pascal to System interface.

SYNOPSIS

#include "/mntb/3/srg/saga/include/system.h" pc * saga/lib/system/system/system.o

DESCRIPTION

The purpose of these routines is to provide a standard interface from Pascal (the pc compiler) to the Unix system. The idea is that the SYS library should be the only thing which needs to be altered to port the Pascal portion of SAGA to System 5 or Xenix (I know, fat chance). There are two essential differences between the Pascal and C versions of the system calls. First strings in Pascal are passed as "systring", and converted to the C NULL terminated format internally. Second pointers in Pascal must be typed. If the **value** of a pointer is required then the "ord()" of that pointer returns an integer which agrees with the type **address** defined in system.h. Sadly, there is not a well defined mechanism for going the other way. An undiscriminated variant record is necessary to convert pointers to integers. Further, the **size** of a record must be calculated by calling a "Delta" function with two var parameters which are successive array elements. The function must be written in C and should define the arguements as integers. For Example:

```
function DeltaMyType(var lo, hi : MyType) : integer ;
                external ;


int
DeltaMyType(lo, hi)
int     lo, hi ;
{
        return(hi - lo) ;
}
```

There are some other special types. The Unix file system sets permission codes for files. In the header files these parameters are always called **mode.** The constants **OtherExec, OtherWrite, .., GroupExec, .., OwnWrite,** can be added together to form the desired permission code. The SYSaccess function has the testmode argument, which takes a sum of the **AccessExist, AccessExec, AccessWrite,** and **AccessRead** constants. The SYSlseek function uses the **SeekAbsolute, SeekRelative,** and **SeekFromEnd** constants (not added together). Finally, the SYSopen function uses the constants **OpenReadOnly, OpenWriteOnly, OpenReadWrite, OpenNoDelay, OpenAppend, OpenCreat, OpenTrunc,** and **OpenExcl.**

Normally the parameters of each SYS procedure correspond to the parameters of the C function. The acceptions are the memory allocation routines, which return the pointer as a var parameter rather than as a function result. Note: these procedures also had to be integrated into the Pascal runtime environment, care should be taken when rewriting.

DIAGNOSTICS

Generally, error returns are the same as for C. SYSerror can be used to obtain a text description of each error, providing there are no intervening SYS calls.

FILES

$

SEE ALSO

Associated C functions, and section 2 introduction.

IDENTIFICATION

George McA Beshers, UIUC DCL Urbana Ill. 61801.

BUGS

The systring type is currently limited to 126 characters which is somewhat small.

NAME

AllocPermid

SYNOPSIS

```
AllocPermid(
        name:        systring) : sypermidindex;
```

DESCRIPTION

This procedure allocates a permanent id for SAGA string and symbol tables. For this routine to work the environment variable SAGA_INDEX_FILE must be set the pathname of a writeable file. The file is maintained in a format similiar to /etc/passwd. Specifically, the permanent id, colon, and the full path name. Unfortunately, AllocPermid is no smarter than **csh**, i.e., it is fooled by symbolic links.

In practice this function need only be called when a new file is created. If the full path name equals one already in the table, that permanent id is returned. Currently, the table size is 1k, the goal being support SAGA (editor, olorin, filters, ...) under SAGA. Another way to think of this is that the SAGA_INDEX_FILE is a view of the SAGA system.

If an error occures a message is printed. Index 1024 is the error return.

DIAGNOSTICS

getwd failed.
Unix United not supported (path starts with /../).
getenv failed (SAGA_INDEX_FILE is not set).
SAGA Index File open failed.

FILES

File specified by SAGA_INDEX_FILE.

SEE ALSO

String.3, Richards Thesis.

IDENTIFICATION

Beshers, George. beshers@uiucdcs.

BUGS

Perhaps one should be the error return. One is a valid permanent id, thus the editor would keep working in an improper environment.

NAME

String Manager – String table management for SAGA.

SYNOPSIS

****** String Table Routines ******

```
createstringtable(
                name:          systring;
                permid:        sypermidindex;
                mode:          integer;
            var rootcontext:   contexttag;
            var error:         boolean);

openstringtable(
                name:          systring;
            var permid:        sypermidindex;
            var rootcontext:   contexttag;
            var error:         boolean);

closestringtable(
                rootcontext:   contexttag;
            var error:         boolean);


flushstringtable(
                rootcontext:   contexttag;
            var error:         boolean);

geterrorflags(
            var errorflags:    errorset);

geterrtext(
                errortype:             syerrorkind;
            var errtxt:        systring);

initstringmanager;


****** String Manipulation Routines ******

insertstring(
                name:          systring;
                context:       contexttag;
            var newstring:     stringtag;
            var found:         boolean;
            var error:         boolean);

retrievestring(
                string:        stringtag;
            var name:          systring;
            var error:         boolean);

locatestring(
```

```
                name:          systring;
                context:       contexttag;
            var string:        stringtag;
            var found:         boolean;
            var error:         boolean);

retrievestringlength(
                string:        stringtag;
            var error:         boolean) : integer;

deletestring(          *Not Active*
                string:        stringtag;
            var error:         boolean);

comparestring(
                strtg1:        stringtag;
                strtg2:        stringtag;
            var error:         boolean) : sycompareresult;

comparestringbystring(
                str1:          systring;
                strtg2:        stringtag;
            var error:         boolean) : sycompareresult;

getstringtype(
                string:        stringtag;
            var stringtype:    integer;
            var error:         boolean);

setstringtype(
                string:        stringtag;
                stringtype:    integer;
            var error:         boolean);

gettagfrag(
                string:        stringtag) : sytagfragment;

buildtag(
                permid:        sypermidindex;
                tagfrag:       sytagfragment) : stringtag;

sycompareresult = (strlt, streq, strgt) ;


****** Systring Utility Routines ******

makestring(
                s :     charbuf ;
            var sy:    systring) ;

concatsystring(
            var result:        systring;
```

```
                first:          systring;
                second:         systring) ;

    int2string(
                i : integer;
            var result:         systring;

    wrsystr(
            var out :           text;
            s : systring) ;
```

DESCRIPTION

These routines constitute the SAGA string manager. The **initstringmanager** routine must be called first since it is the Pascal "solution" to compile time initialization.

The **openstringtable, createstringtable, flushstringtable,** and **closestringtable** procedures provide the file system level access to a string table. The file system procedures append ".str" to the name provided and attempt the operation implied by their name. You can not open or creat the same file (by path name) twice, or two files with the same permanent id. All four of the operations can fail due to file system access failure.

The concept of "contexttag" pertains more to the symbol manager than the string manager, and is used here for compatability. The context tags actually used may be either the root context returned by the **createstringtable** and **openstringtable,** or any other active context for the symbol table with the same permanent id. The permanent id is used to distinguish between different string tables. It is encoded in both "contexttags" and "stringttags" so that a tag uniquely identifiers a particular string throughout the system. The mechanism for assigning permanent ids is described in AllocPermid.

The string manager deals with **systring(s)** which are a record with the following fields:

```
start:  1..126;
count:  0..126;
chars:  array [1..126] of char;
```

Thus if the chars contains "This is a test", with start=4 and count=5 then the string equals "s is ". The procedures makestring, concatsystring, int2systring, and wrsystr are auxilary routines to help manipulate systrings. Note: makestring('testing 1 2 3', s) works fine, but trailing spaces are lost. **Wrsystr** writes the string to the specified file.

The **insertstring** is the only way to put strings into the symbol table. The inserted string's tag is returned in new string. NOTE: if the string exists **found** is set, and NO error is generated, contrary to earlier versions. The **retrievestring** routine is the inverse. It is of course an error to try to retrieve a string associated with an un-opened string table, or a string which doesn't exist. The **retrievestringlength** is faster than retrievestring, used mostly by the editor for screen refresh. The **deletestring** procedure exists, but is disabled because it is not possible to inhibit copying of editor pointers. The **getstringtype** and **setstringtype** permit an integer to be stored with each string for classification purposes (reserved words, function/procedure/variable classification ...).

The **geterrorflags** and **geterrortext** routines are used by both the string and symbol table managers. They should be called whenever the "error" parameter is set upon procedure return.

The **gettagfrag** and **buildtag** routines provide support for optimizations used by the editor. The sytagfragment is a 2 byte quantity, and the stringtag is 4 bytes. This saves some space in the parse tree node.

DIAGNOSTICS

FILES

name.str

SEE ALSO

symbol(3), AllocPermid(3)

IDENTIFICATION

beshers@uiucdcs

BUGS

126 is too small.

SAGA BIBLIOGRAPHY

Roy H. Campbell

Department of Computer Science

University of Illinois at Urbana-Champaign

Urbana, Illinois

July, 1985

SAGA Bibliography

July 1985

1. Badger, W. H., "MAKE: A Seperate Compilation Facility for the SAGA Environment," M.S. Thesis, Dept. of Comp. Sci., Univ. of Illinois at Urbana–Champaign, 1984.
2. Beshers, G. M. and R. H. Campbell, "Maintained and Constructor Attributes," In: The Proceedings of the Symposium on Language Issues in Programming Environments, Seattle, W., pp.34–42, June 1985.
3. Campbell, R. H. and P. Richards, "SAGA: A System to Automate the Management of Software Production," Proceedings of 1981 National Computer Conference, Chicago IL, pp.231–234, May 1985.
4. Campbell, R. H. and P. A. Kirslis, "The SAGA Project: A System for Software Development," Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Software Engineering Notes, Vol. 9.,No. 3, SIGPLAN Notices, Vol 19, No. 5, pp.73–80, May 1984.
5. Campbell, R. H. and P. E. Lauer, "RECIPE: Requirements for an Evolutionary Computer–based Information Processing Environment," IEEE Proceedings of the Software Process Workshop, Egham, Surrey, pp.67–75, February 1984.
6. Dever, S., "A Multi–Language Syntax–Directed Editor," M.S. Thesis, Dept. of Comp. Sci., Univ. of Illinois at Urbana–Champaign, 1981.
7. Essick, IV R. B., "Notesfiles: A UNIX Communication Tool," M.S. Thesis, Department of Comp. Sci. Technical Report #1165, University of Illinois at Urbana–Champaign, Urbana, Illinois, March 1984.
8. Essick, IV R. B., and R. B. Kolstad, "Notesfile Reference Manual," Department of Comp. Sci. Tech. Report #1081, University of Illinois at Urbana–Champaign Urbana, Illinois, October 1982.
9. Hammerslag, D. H., "TED: A Tree Editor with Applications for Theorem Proving," M.S. Thesis, Dept. of Comp. Sci., Univ. of Illinois at Urbana–Champaign, 1984.
10. Hammerslag D., S. N. Kamin and R. H. Campbell, "Tree–Oriented Interactive Processing with an Application to Theorem–Proving," Submitted to Softfair, 1985.
11. Kimball, J., "PCG: A Prototype Incremental Compilation Facility for the SAGA Environment," M.S. Thesis, Dept. of Comp. Sci., Univ. of Illinois at Urbana–Champaign, 1985.
12. Kirslis, P. A., R. B. Terwilliger and R. H. Campbell, *The SAGA Approach to Large Program Development in an Integrated Modular Environment.* Proceedings of the GTE Workshop on Software Engineering Environments for Programming–in–the–Large, pp.44–53, June 1985.
13. Richards, P., *A Prototype Symbol Table Manager for the SAGA Environment,* Master's Thesis, Dept. of Comp. Sci., University of Illinois at Urbana–Champaign, 1984.
14. Terwilliger, R. B. and R. H. Campbell, *ENCOMPASS: a SAGA Based Environment for the Composition fo Programs and Specifications.* Submitted to 19th Hawaii International Conference on System Science (January 1986).

A RANDOM ACCESS FILE I/O PACKAGE FOR PASCAL

Peter A. Kirslis

Department of Computer Science

University of Illinois at Urbana-Champaign

Urbana, Illinois

July, 1985

A Random Access File I/O Package for Pascal

Peter A. Kirslis

July, 1985

1. Introduction

Pascal provides no mechanism to support random access to files. This document describes a software package which permits a program written in Berkeley Pascal to randomly access records in a file. The programmer specifies a record to be paged and provides a buffer (an array of records) to contain a portion of the file in memory. The package of paging routines then provides an interface by which the records in this file can be accessed and modified. Only a small portion of the file needs to be memory resident at any time; the package implements a demand–pager to move the data in and out of memory as required. The routines in the package can also be used to define an interface to treat the records as an encapsulated data type, and implement additional access routines to provide access to the fields in the record in an imlementation independent manner.

2. The Paging System

The paging system provides access to a potentially large file of records through a possibly small area of memory available to a program. Conceptually, the file may be thought of as an array of records, the first one labelled with index 1, and with no upper bound. As higher and higher indices are referenced, additional pages are added to the file. The file is limited in size only by UNIX system imposed restrictions

(typically the amount of free space on the file system containing the file).

Each record in this file can be read or written independently from all others in the file, in any order whatsoever. The programmer using the paging system simply specifies the index of the record in the file he wishes to access, and the record will be swapped into memory if not already present, and made available to him. Figure 1 illustrates both the concept and the implementation scheme used by the routines.

The records to be paged can be any size up to but not greater than the size of the disk page which is swapped by the operating system. On older systems, this size is typically 512 bytes, although page sizes of 1024, 4096, and 8192 bytes are also common.

Since all disk i/o is performed a page at a time, no record is stored across two pages, since this doubles the overhead to retrieve the record. So as many records as will fit onto a single page are stored on that page, and the remaining space is left as a "hole", which is not used by the paging system. This can be seen in the disk file diagram in Figure 1.

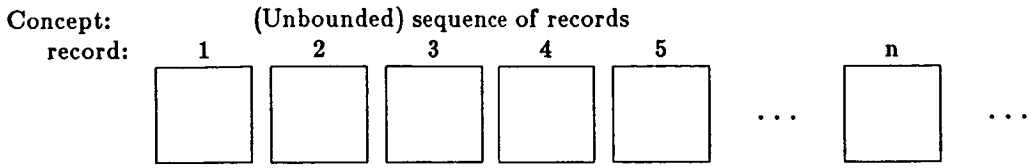
The data is stored in memory as an array of records. The user's program must contain a declaration of the record, and a pointer to an array of records to be used as a buffer to contain the pages of records which will be swapped into and out of memory by the paging system. The routines use a page table and buffer table to store the information needed to manage the data. This information is hidden from the user, and it is not necessary to understand these structures in order to use the paging routines; these structures are shown in Figure 1 only for completeness and the interest of the reader.

3. The Paging Routines

This section presents the declarations of the paging routines. Figure 2 presents a flowgraph illustrating the permissible calling sequences of these routines, to help the reader understand to relationships among the routines.

3.1. Initialization

The first routine called must be *pginit*, to initialize the internal data structures to be used by the pager. The parameters to this routine permit a debugging file to be specified, into which a trace of the



Implementation:

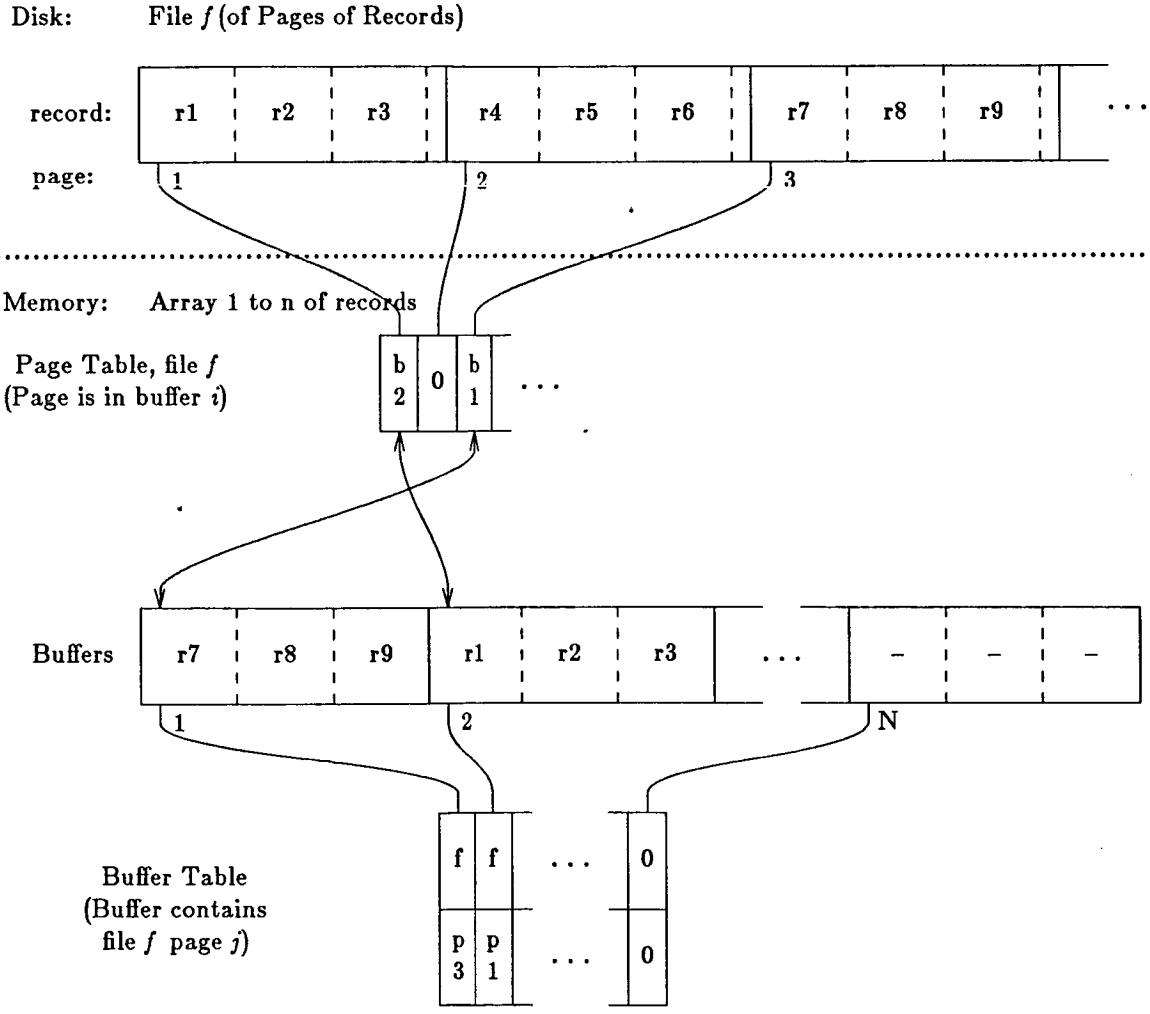


Figure 1: The SAGA Editor Paged File

paging activity will be placed. The paging routines must also have been compiled with the -DDEBUG command line option (or a `#define DEBUG`) line in the source file, so that the debugging code will be included in the object library. A number of debugging levels are possible; these are as follows:

```
Caution: debugflag > 2 produces volumes of output!!!

0 => No debugging (no debug file is created).
1 => Log all routine calls, and each data page swapped between disk and memory,
2 => and also show buffer pool entries after 'pgusebuffer' calls,
3 => and also show every 'pgaccess' call,
4 => and also show page/buffer tables for buffer assignments,
5 => and also show page/buffer tables for buffer releases.
```

```
function pginit              (* Initialize data structures *)
    (debugflag: integer;     (* Nonzero => use debug file *)
     var dfile: charbuf      (* Name of debugfile or 0 *)
    ): integer; external;    (* Return: 0 for success; -1 for error *)
```

3.2. Error Messages

The paging routines do their work silently. If an error occurs, an error code (usually -1) will be returned. A descriptive error message can also be retrieved for printing, if desired, through a call to *pgerror*. This routine returns an error message corresponding to the most recent paging system error encountered.

```
procedure pgerror            (* Get description of most recent error *)
    (var errmsg: charbuf;    (* Return: the error message *)
     var errlen: cbufindex   (* Return: the length of the message *)
    ); external;
```

3.3. File Management

Several routines are provided to manage the files created by the pager: *pgfilecreate, pgfileopen, pgfileclose, pgfileflush, pgfilechmod,* and *pgfiledelete.*

When initially created, a file is assigned access permissions as specified by the *mode* parameter; these permissions can later be changed with *pgfilechmod.* For file creation, the *recsize* parameter must specify the size of the record in bytes. This information is used to determine how many records will fit onto a disk page. The routine returns the number of records per page, which can subsequently be used in a call to

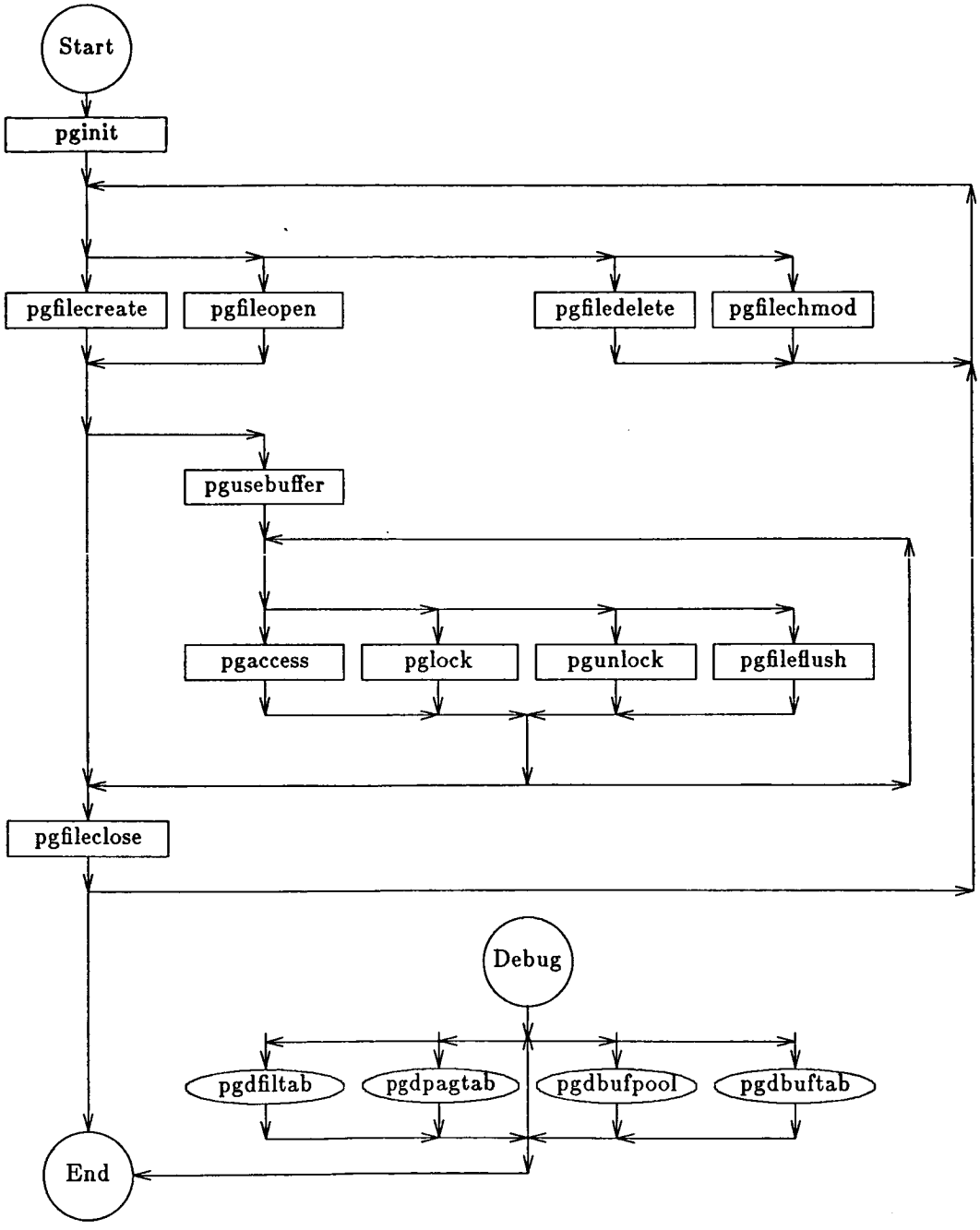


Figure 2: Paging Routine Flowgraph. This figure shows the necessary ordering of calls to the paging routines. All paging routines are shown as rectangles; pager debugging routines are shown as ellipses.

pgusebuffer (described later) to assign a buffer pool to the file. When a file is opened, both the record size and number of records per page are returned to the caller for similar use.

```
function pgfilecreate       (* Create a new paged file of records *)
    (var name: charbuf;     (* Zero-terminated file name *)
     mode: integer;         (* File protection to be assigned *)
     recsize: integer;      (* How many bytes of space per record *)
     var recperpagep:
           integer          (* Return: Number of records per i/o block *)
    ): integer; external;   (* Return: filetag for success, -1 for error *)


function pgfileopen         (* Open an existing record file *)
    (var name: charbuf;     (* Zero-terminated file name *)
     var recsizep: integer; (* Return: number of bytes space per record *)
     var recperpagep:
           integer          (* Return: number of records per i/o block *)
    ): integer; external;   (* Return: filetag for success, -1 for error *)
```

When finished, the program must close the file before exiting. This is necessary to write any data remaining in memory to disk, write the file trailer record (which contains the record and page size of the file), and close the file.

If the programmer wants to periodically write all data that is in memory out to disk without releasing the space and closing the file, then *pgfileflush* should be called.

```
function pgfileclose        (* Write data and close file *)
    (filetag: integer       (* File to write and close *)
    ): integer; external;   (* Return: 0 for success, -1 for error *)


function pgfileflush        (* Flush data to disk, keeping copy in memory *)
    (filetag: integer       (* File to flush *)
    ): integer; external;   (* Return: 0 for success, -1 for error *)
```

File permissions of an existing file can be altered with a call to *pgfilechmod*. A file may be deleted with a call to *pgfiledelete*.

```
function pgfilechmod        (* Change the permissions on the named file *)
    (var path: charbuf;     (* Zero-terminated file name *)
     mode: integer          (* New permissions for file *)
    ): integer; external;   (* Return: 0 for success, -1 for error *)
```

```
function pgfiledelete       (* Delete the named file *)
    (var path: charbuf      (* Zero-terminated file name *)
    ): integer; external;   (* Return: 0 for success, -1 for error *)
```

3.4. Buffer Allocation

A pointer to the data buffer must be declared in Pascal so that the program can reference the records with code written in Pascal. The C routines need the memory address of this buffer so that they can swap the data in and out of memory. But because Pascal is strongly typed, and only one routine is provided for all arrays of records, this pointer must be passed to *pgusebuffer* by first calling *ord(bufptr)* in order to convert all pointers to an integer type which will be accepted by the routine. This value is treated by the routine as a memory address, and used as a reference point when reading and writing the paged data to and from memory.

The *reccount* parameter specifies the size of the buffer: how many records it contains. Since a data buffer which is a fraction of the page size cannot be used, this routine requires the buffer to be an exact multiple of the page size in use. Using the *recperpage* parameter returned by either *pgfilecreate* or *pgfileopen*, the *reccount* parameter of *pgusebuffer* can be set to an integer multiple of this value. A potential difficulty arises since different UNIX file systems on the same computer can be assigned different page sizes, and the program may not be able to find a value that will work for all of them. To protect against this case, the program can first check that its buffer is an exact multiple of the page size, and if it finds that it is not, it can decrease the record count to be passed to *pgusebuffer* to a value that is an exact multiple. There will be some wasted memory at the end of the buffer, but since Pascal does not permit the specification of arrays of records dynamically, there is no other choice.

```
function pgusebuffer        (* Assign a buffer to a file *)
    (filetag: integer;      (* Open file to page in this buffer *)
     bufaddr: integer;      (* Address of buffer: use ord(buffer) *)
     reccount: integer      (* Buffer length in record size *)
    ): integer; external;   (* Return: 0 for success, -1 for error *)
```

3.5. Data Access

Once the file has been created or opened, *pgaccess* is used to access records in the file. This routine expects the *absolute* record index, assuming that the first record in the file is assigned index 1. It takes this index, brings the record into memory if it is not already resident there, and returns a *relative* index into the record in the data buffer where the record may be found. Figure 3 illustrates the mapping that is performed by this routine.

To simplify record access, the programmer is advised to code this function call as follows:

```
buffer^[pgaccess(filetag, recnumber)].field
```

By using this scheme, the program always uses only the absolute record index, while the relative index is used only for addressing to access the actual record. Note that this relative index should not be saved, since subsequent references to the paging routines will eventually cause the page containing this record to be swapped out of memory, probably to be swapped back in later at a different relative index.

Access: Must map (file, record) into (index) in buffer of records:

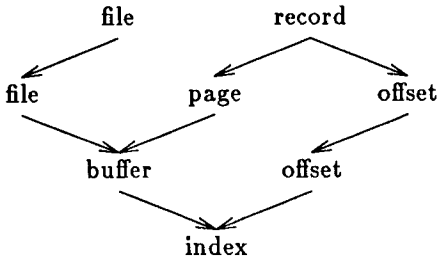


Figure 3: File page to buffer mapping. If the file page is not resident in any buffer, then one must be allocated and the page copied into it. This may cause another page to be written to disk if this buffer was previously in use.

Since this syntax is a bit unwieldy, it is recommended that the programmer provide a function to access each field of the record, thus hiding this implementation information and providing encapsulation of the record reference. A typical function might appear as:

```
function Xfield(recindex: integer): <type-of-field>;
begin
  Xfield := buffer^[pgaccess(filetag, recindex)].field
end;

where X is replaced by some prefix of the programmer's choosing
and <type-of-field> is the data type of the field in the record.
```

Then all references to this field of the record can be written as *Xfield(i)*, a much simpler notation. If many references to fields within a single record are required at once, the form

```
with buffer^[pgaccess(filetag, recindex)] do begin ... end
```

can be used instead to decrease the overhead of all of the *pgaccess* calls which would otherwise be needed.

```
function pgaccess          (* Map a (file, record) to a (buffer) index *)
    (filetag: integer;     (* File containing the record *)
     recnum: integer;      (* Absolute record index *)
     modify: integer       (* Nonzero => this is a write reference *)
    ): integer; external;  (* Return: actual buffer index or 0 for error *)
```

Routines *pglock* and *pgunlock* are a recent addition to the paging system. They are intended to permit the direct use of a pointer to a record over a period of time to decrease the overhead of the paging system. The page containing the record is locked into memory and will not be swapped until all records on that page have been released by *pgunlock*. Caution must be observed if these routines are used, since it is possible to deadlock the system.

A pointer to a record, represented as an integer, is returned in order to permit this routine to be used with many different data types. Since this integer is not acceptable to Pascal, it will need to be type converted into a pointer to a record by a simple C routine which can be written by the programmer. This routine simply returns its argument, but is declared in Pascal as being called with type *integer* and returning a pointer to the desired record. Unless a clear use is seen for this routine, the programmer is advised to use *pgaccess* instead, since it provides the same access.

```
function pglock              (* Lock (file, record) in memory *)
    (filetag: integer;       (* File containing the record *)
     recnum: integer;        (* Absolute record index *)
     modify: integer         (* Nonzero => this is a write reference *)
    ): integer; external;    (* Return: actual memory address of record *)
                             (* or 0 for error *)

function pgunlock            (* Unlock (file, record) from memory *)
    (filetag: integer;       (* File containing the record *)
     recnum: integer         (* Absolute record index *)
    ): integer; external;    (* Return: 0 for success, -1 for error *)
```

4. Debugging Routines

Several additional routines are available which print out the internal data structures used by the paging system. These can be used both to improve the user's understanding of the operatin of the system if desired, and to help track down any bugs that may arise.

Each routine takes a *filetag* parameter corresponding to the file being queried, a *header* parameter to indicate whether a table header should be displayed, and a *where* parameter, which should be set to either 1 or 2 to indicate printing on either standard output or standard error respectively.

```
void
pgd_filtab(filetag, header, where)     /* display filtab structure */
int    filetag;                        /* display for this file */
int    header;                         /* nonzero => print header line */
int    where;                          /* where to print data */

void
pgd_bufpool(filetag, header, where)    /* diplay buffer pool struct */
int    filetag;                        /* display for this file */
int    header;                         /* nonzero => print header line */
int    where;                          /* where to print data */

void
pgd_pagtab(filetag, header, where)     /* display page table */
int    filetag;                        /* display for this file */
int    header;                         /* nonzero => print header line */
int    where;                          /* where to print data */
```

```
void
pgd_buftab(filetag, header, where)      /* display buffer table */
int    filetag;                         /* display for this file */
int    header;                          /* nonzero => print header line */
int    where;                           /* where to print data */
```

5. Using the Paging System Library

The paging system declarations are included in the user's program with the following statement:

```
#include "pager.h"
```

When the program is compiled, the paging routines are linked into the program during the load step of the compilation:

```
pc -c yourprogram.p
pc -o yourprogram yourprogram.o pager.o
```

6. Summary

The paging routines provide a mechanism by which random access file i/o can be performed from Pascal programs, and by which a potentially very large file of data can be accessed in a program using a possibly small amount of memory. The cost of these functions is the increased overhead of a procedure call per record reference. These routines are used by the SAGA language-oriented editor to manage the parse trees which are constructed during the editing process. Questions should be directed to the author.